

THE LOGIC OF ADAPTIVE BEHAVIOR

KNOWLEDGE REPRESENTATION AND ALGORITHMS FOR
ADAPTIVE SEQUENTIAL DECISION MAKING UNDER UNCERTAINTY
IN FIRST-ORDER AND RELATIONAL DOMAINS

Martijn van Otterlo

Bibliographic information

First Version (May 2008)

PhD Thesis – University of Twente – The Netherlands
ISBN 978-90-365-2677-7

Second Version (Feb 2009)

Published and Printed by IOS Press – Amsterdam
ISBN 978-1-58603-969-1

Third Version (May 2010)

Identical to the 2009 version, to be cited as
Martijn van Otterlo (2009) *The Logic of Adaptive Behavior*, IOS Press, Amsterdam

Copyright © 2008, 2009, 2010 by Martijn van Otterlo
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without the written permission of the author.

THE LOGIC OF ADAPTIVE BEHAVIOR

Martijn van Otterlo

Toastmaster

"Gentlemen, pray silence for the President of the Royal Society for Putting Things on Top of Other Things."

Sir William

"I thank you, gentlemen. The year has been a good one for the Society (hear, hear). This year our members have put more things on top of other things than ever before. But, I should warn you, this is no time for complacency. No, there are still many things, and I cannot emphasize this too strongly, not on top of other things. I myself, on my way here this evening, saw a thing that was not on top of another thing in any way. (shame!) Shame indeed but we must not allow ourselves to become too despondent. For, we must never forget that if there was not one thing that was not on top of another thing our society would be nothing more than a meaningless body of men that had gathered together for no good purpose. But we flourish. This year our Australasian members and the various organizations affiliated to our Australasian branches put no fewer than twenty-two things on top of other things. (applause) Well done all of you. But there is one cloud on the horizon. In this last year our Staffordshire branch has not succeeded in putting one thing on top of another (shame!). Therefore I call upon our Staffordshire delegate to explain this weird behaviour."

— *"The Royal Society For Putting Things On Top Of Other Things" sketch, Monty Python's Flying Circus, programme 18 (1970)*

"Er wordt niets nieuws gezegd. Alles wordt nieuw gezegd." (Bomans)

"Es gibt nichts neues, nur neue Kombinationen." (Goethe)

"No one really starts anything new, Mrs. Nemur. Everyone builds on other men's failures. There is nothing really original in science. What each man contributes to the sum of knowledge is what counts." (Flowers for Algernon - D. Keyes, 1966)

Preface

One of my favorite stories is *The Library of Babel* (1941) by the Argentinean writer and librarian *Jorge Luis Borges* (1899–1986). In this fantastic story, Borges describes an imaginary library containing *all possible* books of a specific length, containing a specific number of pages and symbols. The library itself consists of an enormous amount of interconnected hexagonal rooms. Borges describes that “[W]hen it was proclaimed that the Library contained all books, the first impression was one of extravagant happiness. All men felt themselves to be the masters of an intact and secret treasure. There was no personal or world problem whose eloquent solution did not exist in some hexagon.”

For computer scientists and mathematicians, the concept of the universal library immediately activates connections with *combinatorics* and *permutations*. In fact, most computer scientists will be able to program in roughly ten minutes an algorithm to *generate* all possible books (though actually running it to completion will not be possible in their lifetime). The physical concept of the interconnected hexagons has interesting links to *graph theory* and *search spaces*. By all means, if the universal library would really exist, every writer could just ‘look up’, or *search the connection graph* for, the book he or she wants to write, instead of writing it him- or herself. Looking up the right book is, unfortunately, not very easy given that the library is extremely large. Traveling through even the slightest portion of corridors and hexagons will take up more than a lifetime for any mere mortal being. Even stronger, the physical size of the library exceeds the capacity of our universe, as the mathematician Goldbloom Bloch (2008) writes in his exciting book on the library. The labyrinth of the library is thus fictional, yet it is far more intuitive (and fun) to imagine wandering through the physical version. Borges tells us that “[F]or four centuries now men have exhausted the hexagons . . . There are official searchers, inquisitors. I have seen them in the performance of their function: they always arrive extremely tired from their journeys; they speak of a broken stairway which almost killed them; they talk with the librarian of galleries and stairs; sometimes they pick up the nearest volume and leaf through it, looking for infamous words. Obviously, no one expects to discover anything.”

Wandering through a library, not expecting to discover anything is one of the pleasant things in life, I think, although more modern variants exist through the vast network of interconnected webpages on the internet, where hyperlinks correspond to corridors. Umberto Eco was inspired by Borges’ library for his book *The Name of the Rose* (1980), in which he describes an exciting hunt through a physical version of a similar library. Here though, the travelers were looking for something specific, for knowledge, even though it remains unclear what exactly that is but until the end of the story. Eco acknowledged the

inspiration through the librarian in his story, named *Jorge von Burgos*. And although the travelers in Eco's story were looking for general knowledge, the ultimate form of general knowledge in the library has many connections with *compression*, *orderings* and *information theory*: "[W]e also know of another superstition of that time: that of the Man of the Book. On some shelf in some hexagon (men reasoned) there must exist a book which is the formula and perfect compendium of all the rest: some librarian has gone through it and he is analogous to a god. In the language of this zone vestiges of this remote functionary's cult still persist. Many wandered in search of Him."

Out of the many interesting ideas in Borges' story, I find this concept of a book that provides a perfect compendium to all other books most intriguing. The existence of this one book that summarizes all other books in whatever 'best' way is an extraordinary phenomenon. It has to exist though, given that the library is finite. Yet, I find it hard to imagine what the contents would look like. In a much more modest setting, imagining a book that summarizes a well-chosen set of texts on a particular topic is much more conceivable. In fact, this is what I have tried to do in this book: providing a compendium of all work concerning *learning sequential decision making under uncertainty in first-order and relational domains*, in whatever 'best' way possible. I have wandered through a partially physical, partially electronic, Borgian library of literature, discovered new hexagons and books myself, and I have drawn a map of interesting hexagons, corridors, galleries and books that were visited by other travelers.

Research on this topic began in 1998 and has since then continued to grow. It is now an established subfield of AI and results appear at international scientific fora that deal with topics such as *machine learning*, *knowledge representation*, *decision making* and *planning*. Furthermore, it is also an *active* field, and there is a growing attention for its problems and results. Among others, this year the ICAPS 2008 workshop on *a reality check for planning and scheduling under uncertainty* was organized, as well as the AAAI 2008 workshop on *transfer learning for complex tasks*. Both are much related to the core problems in this book. Furthermore, there was a tutorial on *decision-theoretic planning and learning in relational domains* at AAAI 2008, and a tutorial on *first-order planning techniques* at ICAPS 2008. In the coming years, there will be many more scientific events where results concerning sequential decision making in first-order domains will be presented. However, the research in the past decade has already developed a thorough understanding of, and a relatively established core set of techniques for, the Markov decision process setting in first-order and relational domains. It is exactly this topic that I describe in the book. A similar setting, but for non-relational domains, was described a decade ago in the famous book by Sutton and Barto (1998).

Even though I did my ultimate best to cover every text that was relevant for the topic in this book, I cannot possibly guarantee that there are no hidden hexagons that I missed by mistake. In addition, almost all hexagons I describe contain books that are written in English. Though on my journey, I have encountered several texts written in Japanese and Chinese texts for example, that seemed to describe matters that were of interest to me. Unfortunately my knowledge of these languages is limited. Yet, for all English texts I am confident that I might have discovered them all. This book contains a bibliographic map that describes all relevant hexagons. As we will see, there are cases where fellow travelers were not aware of other travelers having discovered interesting hexagons before them, and others had locked themselves inside a particular hexagon without looking for paths that

connected their hexagon with another that contained similar books. As far as possible, I have tried to establish links between hexagons, and books, assembling the knowledge of all travelers that have gone before me, and I have even dug new tunnels connecting hexagons when relevant or necessary.

This book grew out of my recent dissertation (van Otterlo, 2008a). Writing a book like this one has been a long, personal journey through a vast, Borgian library. Fortunately, during this journey, I have met many kind and interesting fellow travelers. Sometimes our paths crossed several times, sometimes I stumbled across them in some remote, dimly lit, hidden hexagon, and some people have watched over me while I was traveling from one hexagon to another. I have had the pleasure to engage in interesting conversations with many of the fellow travelers that I mention in this book. Many talks in person or by e-mail have helped me in increasing my understanding of sometimes complicated matters. A possible list of names would quickly turn into a Borgian space of its own, and thus here I take the opportunity to thank you all at once. I also thank all of my coauthors for working with me on interesting topics. In the recent years, I have had the honor to be a member of the Dutch discussion group on artificial intelligence, EMERGENTIA, and I would like to thank all its members and former members for fun and interesting discussions on many Sunday afternoons.

I owe much to my promotor John-Jules Meyer and assistant-promotor and referent Marco Wiering. Both have watched over my travels from a small distance and contributed to the final success of my journey with their kind, and complementary, support. I also thank all other wise men who agreed to take place in my committee and read my dissertation. I specifically want to thank Luc De Raedt and Joost Kok. Luc, for all the opportunities he has given me in Dagstuhl, Freiburg and Leuven, and also for showing me how science works, and Joost, for giving me the kind opportunity to create this book in this form.

The scientific community has much to offer, but friends and family are most important still. Many thanks go to my parents and my sister and in addition to all friends and people of all kinds of family. Yet, the one I have to – and gratefully want to – thank the most, is my dear Marieke. In addition to love, friendship, irresistible humor, and much support she has given me slightly longer already than the decade I describe in this book, she has always shown a great *understanding* of me and my journey. I admire that, especially given the scope and length of my travels. I'm looking forward to all our travels still to come.

Leuven, November 2008

Martijn van Otterlo

Contents

Preface	iii
CHAPTER 1 Introduction	1
1.1. Science and Engineering of Adaptive Behavior	3
1.1.1. Artificial Intelligence	3
1.1.2. Constructing Artificial Behavior	4
1.1.3. The Reinforcement Learning Paradigm	7
1.2. You Can Only Learn What You Can Represent	8
1.2.1. Generalization, Abstraction and Representation Formation	10
1.2.2. CANTOR: Representing the World in Snapshots	11
1.2.3. BOOLE: Representing the World in Twenty Questions	12
1.2.4. FREGE: Representing the World in Terms of Objects and Relations . .	14
1.2.5. The World Might be Larger than We See	16
1.3. About the Contents and Structure of this Book	17
1.3.1. Main Theme of This Book	17
1.3.2. A Road Map	24
1.3.3. Other Main Themes and Contributions	28
I Learning Sequential Decision Making under Uncertainty	29
CHAPTER 2 Markov Decision Processes: Concepts and Algorithms	31
2.1. Learning Sequential Decision Making	33
2.2. A Formal Framework	38
2.2.1. Markov Decision Processes.	38
2.2.2. Policies	40
2.2.3. Optimality Criteria and Discounting	41
2.3. Value Functions and Bellman Equations	42
2.4. Solving Markov decision processes	44
2.5. Dynamic Programming: Model-Based Solution Techniques	46
2.5.1. Fundamental DP Algorithms	46
2.5.2. Efficient DP Algorithms	50
2.6. Reinforcement Learning: Model-Free Solution Techniques	52
2.6.1. Temporal Difference Learning	54

2.6.2. Monte Carlo Methods	57
2.6.3. Efficient Exploration and Value Updating	58
2.7. Beyond the Markov Assumption	62
2.7.1. Partially Observable Markov Decision Processes	64
2.8. Discussion	66
CHAPTER 3 Generalization and Abstraction in Markov Decision Processes	69
3.1. From Algorithmic to Representational	71
3.1.1. Algorithmic Aspects	73
3.1.2. Fundamental Problems of Huge State Spaces	74
3.1.3. Representational Aspects	75
3.2. The Essence of Abstraction	76
3.2.1. Knowledge Representation	77
3.2.2. Definitions and Theories of Abstraction	79
3.2.3. Representation Change	79
3.3. Abstraction in the MDP Setting	81
3.3.1. Dimensions of MDP Abstractions	83
3.3.2. The PIAGET-Principle	87
3.3.3. Representations in MDP Abstractions	90
3.4. ABSTRACTION TYPE I: State Spaces	92
3.4.1. Model-Based State Abstractions	96
3.4.2. Model-Free State Abstractions	99
3.5. ABSTRACTION TYPE II: Factored Markov Decision Processes	99
3.5.1. Structured Representation	100
3.5.2. Structured Algorithms	102
3.6. ABSTRACTION TYPE III: Value Function Approximation	106
3.6.1. Fundamentals of Value Function Approximation	109
3.6.2. Architectures for VFA	113
3.7. ABSTRACTION TYPE IV: Searching in Policy Space	127
3.8. ABSTRACTION TYPE V: Hierarchical and Temporal Abstraction	129
3.8.1. Semi-Markov Decision Processes	131
3.8.2. Fixed Hierarchical Abstractions	132
3.8.3. Model-Minimization for SMDPs	136
3.8.4. Dynamic Hierarchical Abstractions	137
3.9. An Abstraction Case Study: Fingerprint Recognition	140
3.9.1. Reinforcement Learning for Minutiae Detection	141
3.9.2. Experimental Results	142
3.9.3. Benefits of Various Abstractions	143
3.10. Discussion	144
 II Sequential Decisions in the First-Order Setting	 151
CHAPTER 4 Reasoning, Learning and Acting in Worlds with Objects	153
4.1. The World Consists of Objects	159
4.1.1. Objects are Omnipresent and Indispensable	159
4.1.2. A Relational Domain: BLOCKS WORLD	163

4.1.3. Representing a World of Objects and Relations	165
4.2. Representation and Inference in First-Order Domains	171
4.2.1. First-Order Logic	172
4.2.2. Fragments and Extensions of FOL	177
4.2.3. First-Order Abstraction and Generalization	190
4.3. Learning in First-Order Domains	193
4.3.1. Obtaining Logical Abstractions	193
4.3.2. Inductive Logic Programming	195
4.3.3. Statistical Relational Learning	203
4.4. Acting in First-Order Domains	206
4.4.1. Formalizing and Modeling First-Order Domains	206
4.4.2. Two Characteristic Systems	211
4.4.3. Beyond Basic Action Theories	218
4.5. Learning Sequential Decision Making in Relational Domains	219
4.5.1. Lifting the MDP Framework to First-Order Domains	219
4.5.2. The PIAGET-Principle in First-Order Domains	231
4.5.3. Learning and Representation Tasks in Relational RL	240
4.5.4. What is Relational RL?: Different Viewpoints	242
4.6. Conclusions	243
CHAPTER 5 Model-Free Algorithms for Relational MDPs	247
5.1. Model-Free Relational Reinforcement Learning	248
5.1.1. Sampling and Structural Induction	250
5.1.2. Representations, and Value Functions vs. Policies	251
5.2. CARCASS: A Model-Free, Value-Based Approach	252
5.2.1. Relational Abstractions over RMDPs	252
5.2.2. Q -Learning for CARCASSs	257
5.2.3. Indirect Value Learning for CARCASSs using Approximate Models	259
5.2.4. Analysis and Experiments	261
5.2.5. Discussion	270
5.3. A Survey of Model-Free, Value-Based Approaches	271
5.3.1. Value-Based Learning on Fixed Abstraction Levels	271
5.3.2. Value-Based Learning using Dynamic Generalization	276
5.3.3. Discussion of Model-Free, Value-Based Techniques	283
5.4. GREY: Evolutionary Policy Search in Relational Domains	285
5.4.1. Evolutionary Search and ILP	285
5.4.2. GREY's Anatomy	287
5.4.3. Experimental Evaluation	291
5.5. A Survey of Policy-Based Model-Free Relational RL	297
5.5.1. Evolutionary Policy Search	297
5.5.2. Policy Search as Classification	298
5.5.3. Policy Gradient Approaches	300
5.6. Discussion	301
CHAPTER 6 Model-Based Algorithms for Relational MDPs	305
6.1. Intensional Dynamic Programming in Five Easy Steps	309
6.1.1. STEP I: Classical Dynamic Programming	310

CONTENTS

6.1.2.	STEP II: Replacing Tables by Sets	313
6.1.3.	STEP III: Set-Based Value Functions	317
6.1.4.	STEP IV: Set-Based Dynamic Programming	321
6.1.5.	STEP V: Intensional Dynamic Programming	324
6.2.	A Relational State Description Language	347
6.2.1.	Abstract States	347
6.2.2.	Abstract Actions	349
6.2.3.	Rewards	350
6.2.4.	Domain Theory and Constraint Handling	351
6.2.5.	Markov Decision Programs, Value Functions and Policies	353
6.3.	REBEL: Value Iteration for Markov Decision Programs	355
6.3.1.	Overlaps, Regression and Weakest Preconditions	356
6.3.2.	Combination and First-Order Decision-Theoretic Regression	361
6.3.3.	Maximization: Computing Abstract State Values	364
6.3.4.	Relational Bellman Backup Operator	365
6.3.5.	Experiments	366
6.4.	Logic Programming meets Dynamic Programming	374
6.4.1.	Tabling	374
6.4.2.	Policy Induction in REBEL	376
6.4.3.	Other Extensions and Domain Theories	379
6.5.	A Survey of Model-Based Approaches	381
6.5.1.	Methods for exact IDP in First-Order Domains	381
6.5.2.	Approximate Model-Based Methods for First-Order MDPs	386
6.5.3.	Beyond the Markov Assumption	391
6.6.	Discussion	392

III Implications, Challenges and Conclusions 395

CHAPTER 7	Sapience, Models and Hierarchy	397
7.1.	Scaling Up	399
7.1.1.	Extending Mental States	399
7.1.2.	Declarative versus Procedural Representations	400
7.1.3.	Learning vs. Reasoning	401
7.1.4.	Examples of Existing Formalisms	402
7.2.	Characterizing Sapient Agents	403
7.2.1.	Cognitive Agents	404
7.2.2.	Learning in Cognitive Agents	407
7.2.3.	The Social Environment	409
7.2.4.	Discussion of the Sapient Model of Agents	410
7.3.	A Survey of Hierarchies, Models, Guidance and Transfer	412
7.3.1.	Learning World Models	412
7.3.2.	Bias, Guidance and Heuristics	415
7.3.3.	Hierarchies	416
7.3.4.	Transfer	417
7.3.5.	Multi-Agent Approaches	419

7.4. Discussion	420
CHAPTER 8 Conclusions and Future Directions	423
8.1. Conclusions and Reflections	424
8.1.1. Main Argument	424
8.1.2. Contributions	428
8.1.3. Dimensions of First-Order MDPs and Solution Algorithms	429
8.2. Future Challenges	433
8.2.1. Upgrading the Complete Spectrum of RL Methods	433
8.2.2. Techniques Developed in this Book	433
8.2.3. Representational Aspects	435
8.2.4. Algorithmic Aspects	436
8.2.5. Theory	436
8.2.6. Agents, Cognitive Architectures, Reasoning and Transfer	439
8.2.7. Applications, Actions, Robots and Activities	439
8.2.8. Benchmarks and Toolboxes	441
8.2.9. Beyond the Markov Assumption	441
8.3. Concluding Remarks	442
Bibliography	443
List of Acronyms	477
Author Index	479

CONTENTS

CHAPTER 1

Introduction

DECISION MAKING IS A VERY CHALLENGING PROBLEM, both in human thinking as in *artificial intelligence* systems. While you are reading this text, many things take place inside your brain. For one thing, you are trying to stay focused on reading this, you are trying to keep yourself nourished, you are trying to remember to send this very important *e-mail*, and so on. Furthermore, you know how to ride a bicycle, you know how to make coffee and you may know how to write a report using \LaTeX , and many more such things. And, additionally, you may have knowledge about Bayesian networks, your left ear, table spoons and possibly even about ninja swords. How on earth can you possibly decide on your next action?

Apparently, humans have the ability to store many types of knowledge, operational skills, and do many types of reasoning processes, all at the same time. A complete explanation of this phenomenon, and a working computer-based implementation of such processes, counts as the *Holy Grail* of the field of *artificial intelligence*. Therefore, let us first take a look at the significantly more restricted setting of decision making in Figure 1.1. These examples were described by Tversky and Kahneman (1981), who experimented with variants of essentially the same decision problem and investigated the influence of how people interpret the problem on their decisions. The variance in the answer distribution in the two problems is explained by the authors as

”The majority choice in this problem is risk averse: the prospect of certainly saving 200 lives is more attractive than a risky prospect of equal expected value, that is, a one-in-three chance of saving 600 lives. [...] The majority choice in problem 2 is risk taking: the certain death of 400 people is less acceptable than the two-in-three chance that 600 will die. The preferences in problems 1 and 2 illustrate a common pattern: choices involving gains are often risk averse and choices involving losses are often risk taking. However, it is easy to see that the two problems are effectively identical.”

Interestingly, for humans it seems to matter how a particular problem is *represented*. Both problems pose the same dilemma, but trigger different responses, due to a concept called *decision frame* that refers to the decision-maker’s conception of the acts, outcomes, and contingencies associated with a particular choice.

From this example, we see that the representation of a decision problem can be just as important as the intrinsic difficulty of making the decision itself. On the contrary, for

Imagine that the U.S. is preparing for the outbreak of an unusual Asian disease, which is expected to kill 600 people. Two alternative programs to combat the disease have been proposed. Assume that the exact scientific estimate of the consequences of the programs are as follows:

Problem 1 [N=152]:

If Program C is adopted 400 people will die [22 percent]

If Program D is adopted there is a $\frac{1}{3}$ probability that nobody will die, and $\frac{2}{3}$ probability that 600 people will die [78 percent]

Problem 2 [N=155]

If Program A is adopted, 200 people will be saved. [72 percent]

If Program B is adopted, there is a $\frac{1}{3}$ probability that 600 people will be saved, and $\frac{2}{3}$ probability that no people will be saved [28 percent]

Figure 1.1: A deceiving decision problem for humans. N is the number of people in the survey, and bracketed numbers in the answers denote what percentage of respondents chose a particular answer.

a computer, the representation of a problem is relatively meaningless. As long as all the necessary information is present, and it knows how the answer can be computed in whatever mechanical way, correct answers can be ensured, i.e. computers are *rational* entities (Russell, 1997). Still they *are* heavily dependent on representation, albeit in a different way. It does not influence the correctness of the computer's decisions, but it does influence the range of problems they can solve, the generality of their solutions and furthermore how efficient solutions are computed. This is the main theme in this book.

Now, deciding a single thing to do may already be challenging and dependent on representation. But let us go one step further, to *sequential* decision making. Consider the game of CHESS. Each move in a CHESS game is important, but it is the complete game consisting of around 40 consecutive moves that determines winning or losing. Playing a bad move might not be too disastrous for winning the game in the end, though this also depends on the opponent. To play a game of CHESS successfully requires one to plan ahead, and to cope with possibly unforeseen circumstances along the way. This is also influenced by uncertainty about your opponent's moves, which can make your planned strategy fail and force you to adjust.

Whereas computers can be programmed to play games like CHESS, or to perform other sequential decision making tasks such as navigating a robot in a factory, ultimately we would like an intelligent system to *learn* these things by itself. When humans learn how to play CHESS, they use a variety of learning techniques to master the game. Initially, they have to be taught the rules of the game, but after that, they usually acquire increasing levels of play by *practicing* the game, and observing the effects of moves on the outcome of the game. People are not told the optimal moves for each possible CHESS board, but they *learn to evaluate* moves and positions, in order to play better moves. Furthermore, they *generalize* what they have learned such that such knowledge can be applied in 'similar' situations or when playing against 'similar' opponents. In this book we study computer algorithms that mimic this type of learning in sequential decision making tasks. A central role is played by the *representation* of such problems, because these determine which types of problems can be learned (see Section 1.2 for an extended example).

Summarizing, this book is about **learning** behaviors for **sequential decision making tasks** in which there is a significant amount of **uncertainty** and **limited, delayed feedback**. The core topic is about employing **first-order knowledge representation** in such tasks, which is a particular way to 'see' the problem in terms of *objects* and *relations* between objects. The purpose of this introductory chapter is threefold. In the first place, its intention is to introduce the reader to the topic and focus of the research as described in this book. Taking a helicopter view, the location of the matter can be found by zooming in on the field of *artificial intelligence*, then on *machine learning* and finally on *reinforcement learning*. The exact focus of the research is concerned with the *representational aspects* of the reinforcement learning methodology, and in particular the use of powerful *first-order* or *relational* representational devices. The second goal of this chapter is to provide a road map through the chapters of this book. The final aim of the chapter is to highlight the contributions of this book, and their embedding in an existing body of research as reported in the literature.

1.1. Science and Engineering of Adaptive Behavior

We are interested in creating artificial behaviors for sequential decision making. More specifically, we are interested in artificial systems that *learn* how to *do* something. The field of *artificial intelligence* (AI) has been studying this for decades, taking inspiration from many different fields.

1.1.1 Artificial Intelligence

AI is a large field of research that tries to build *systems that perform tasks in which it is understood that some form of intelligence is required*. AI is generally seen as a subfield of computer science, but its connections with and influences from other fields are much more diverse and include *cognitive science, engineering, psychology, biology, sociology, economics, philosophy* and *mathematics*. Many books exist that provide general treatments of the field (Nilsson, 1980; Görtz *et al.*, 2003; Luger, 2002; Russell and Norvig, 2003) of which some are more *logically* oriented (Poole *et al.*, 1998; Minker, 2000b), others deal with *embodied* views on AI (Pfeifer and Scheier, 1999), and yet others deal with the conceptual ideas of AI (Hofstadter, 1979; Minsky, 1985; Haugeland, 1997; Baum, 2004).

AI was originally founded in 1956 and has been occupied with studying, and building *minds* (Haugeland, 1997). An exact characterization of *intelligence* is not all that important to understanding it. Whether some system is intelligent will always be debatable, and therefore, the important question is the following. Given some behavior (by e.g. a human or an animal) that we find interesting in some way, how does this behavior come about? Many sub-fields in AI have developed based on this question, studying various topics such as *memory, vision, logical and commonsense reasoning, navigation, physical movement, evolution, brain functioning*, and most importantly for this book, *decision making and learning*. Much has been achieved so far, also witnessed by widespread use of *expert systems, datamining*, and even *fuzzy controllers* in washing machines and much has still to come¹.

¹Predictions about the future of AI trigger many sorts of reactions, and are often disproved later. For example Hofstadter (1979, p.678)'s predictions on the possibility of a computer beating anyone with CHESS was disproved by the victory of *Deep Blue* over the best human player Gary Kasparov (Schaeffer and Plaat, 1997). Other well-known predictions on whether a robot team will beat the human best team at soccer in

Since the eighties, AI has developed into a strong discipline of science, embracing approaches from other fields such as control theory, statistics, mathematics and operations research, and supported by theories, rigorous experiments, and applications. Owing much to the work by Pearl (1988), AI is now dominated by *probabilistic* approaches. In the mid-nineties, the *agent* metaphor (Wooldridge and Jennings, 1995) became popular as a core object of study (Russell and Norvig, 2003) and nowadays the *game industry* – which dominates the movie industry in terms of financial investments – has discovered AI as a way to make their products *smarter*².

An important dichotomy in AI is that between *general-purpose systems* and *performance systems* (see Nilsson, 1995, 2005, for further discussion). The first is about the systems AI basically started out with; those that aim at understanding and building general, human-like intelligent systems. The second is about programs that are highly specialized and limited to a particular area of expertise. It is related to an old, yet persistent, debate in AI between *strong* AI in which the appropriately programmed computer is really considered a mind, and *weak* AI, in which the principal value of a computer is to be a very powerful tool to formulate and test hypotheses in a rigorous and precise fashion. Many of the current AI approaches belong to the latter category, causing AI to be subdivided into a large number of nearly disjoint fields, for example logical inference vs. probabilistic inference, empirical vs. purely theoretic approaches, and many more fine-grained subdivisions. It includes the work in this book, which is targeted at a very specific area, that of learning sequential decision making. Yet, we argue that the best way is to pursue research into such individual subdivisions, while keeping in mind the needs and constraints of general AI architectures. Or, so to say, *keeping the eye on the prize* (Nilsson, 1995).

1.1.2 Constructing Artificial Behavior

AI has produced several distinct ways to build intelligent agents that can perform well in sequential decision making problems under uncertainty. Note that we focus here on *reactive behaviors* in which the agent's main task is to choose an action based on its current state. In general, we can distinguish three main types of approaches to obtain a controller for the robot's actions, which are *programming*, *planning or reasoning*, and *learning*.

1.1.2.1 PROGRAMMING

The first thing that comes to mind when creating an agent for a specific task is to write a program that completely drives the agent's behavior. The advantages are that the behavior can be tested, it can be set up and programmed in a modular way, and that guarantees can be given about its performance. However, for most realistic problems this is impossible to do. There can be uncertainty about the environment's dynamics, about possible effects of actions, about behaviors of possible other agents in the environment and so on. Furthermore, some aspects of the environment may be *inaccessible* to the agent, such that it misses vital information for its current decision. In addition, programmed behaviors are not robust to changes in the environment, or unforeseen circumstances. In other words, programmed systems are often *brittle* (Holland, 1986), and adaptive systems are preferred.

2050, and whether robots will dominate humans in the near future, remain to be seen.

²When graphical techniques were still developing, games would advertise with increasingly better looking graphics. Nowadays they advertise with slogans such as "*Enhanced AI opponents included*".

1.1.2.2 REASONING AND PLANNING

Instead of fixing the complete behavior beforehand by programming, a second option is to supply all information about the environment to the agent and let it *reason* about it to *plan* ahead a suitable course of action. In deterministic environments this is very well possible, though in environments with uncertainty about the outcomes of actions it becomes more challenging because there are no guarantees that the current plan will reach the goal. On the other hand, giving the agent the ability to plan enables it to cope with such circumstances, for example by adjusting the plan when needed.

For planning to work, the agent must first know everything about the domain. This includes facts, for example that room_1 is also known as *the coffee room*, but also knowledge about how certain things in the environment change either because of the agent's actions or because of external factors. The main challenge is to make this knowledge as complete and as precise as possible. Haddon (2003) tells the story of a fifteen year old boy named Christopher Boon who has Asperger's Syndrome, and in many ways Christopher requires the same kind of precision that is required for a computer.

"And this is because when people tell you what to do it is usually confusing and does not make sense. For example, people often say 'Be quiet', but they don't tell you how long to be quiet for. Or you see a sign which says KEEP OFF THE GRASS but it should say KEEP OFF THE GRASS AROUND THIS SIGN or KEEP OFF THE GRASS IN THIS PARK because there is lots of grass you are allowed to walk on." (Haddon, 2003, p.38).

Although complete and precise formalizations are required, there is a delicate trade-off with the employment of this knowledge in an actual reasoning system. Because computers lack a kind of *commonsense reasoning*, they cannot naturally distinguish between *relevant* and *irrelevant* lines of reasoning. For example, Dennett (1998) describes a robot that spends all its time reasoning about the possible consequences of its actions, without actually *doing* anything anymore. Thus, in addition to knowledge, for planning to work the agent must have efficient reasoning mechanisms that use the information wisely. Otherwise it might end up thinking about (or even doing) stupid things, like Christopher.

"Stupid things are things like emptying a jar of peanut butter onto the table in the kitchen and making it level with a knife so it covers all the table right to the edges, or burning things on the gas stove to see what happened to them, like my shoes or silver foil or sugar." (Haddon, 2003, p.60).

Planning approaches are widespread, and in Section 1.3.1.3 we will briefly outline some historical developments. Most of these approaches cannot be employed in domains with significant uncertainty, and are impossible to apply when information about the dynamics of the domain is absent.

1.1.2.3 LEARNING

In the context of uncertainty and the inability of specifying all necessary information beforehand, it would be best to supply the agent with all the information that is available, and let it *learn* from experience how to perform the task. In other words, *"learning is more*

economical than genetically³ prewiring” (Minsky, 1985, Section 11.7). *Machine learning* (ML) (Mitchell, 1997) is a large sub-field of AI and it deals with various kinds of learning, or *adaptive systems*. A general definition of learning is *the process or technique by which a device modifies its own behavior as the result of its past experience and performance*.

Learning algorithms can be classified along several dimensions, which include the *type* of problem (e.g. classification, behavior), the *knowledge representation* used (see more on this later), and the *source* of the learning experiences. Examples of the latter include datasets and simulation environments, but also prior knowledge that may be available about the domain. One of the most important dimensions in ML algorithms is the *amount of feedback* that is available to the learning system. Basically, there are three types of amounts, ranging from full feedback to essentially none.

Supervised learning is the most common form of ML. Usually the desired result is a *mapping* from problem instances to a set of *class values*. A *training set* that contains *examples* of problem instances along with their desired class label are given to the system. The task now is to take the training set and use it to construct a *generalized* mapping that can label the instances correctly, but in addition, that can label other, *unseen* examples correctly too. An example of such a problem can be found in *direct marketing*. Let us assume a company has much information about its customers, for example buying habits, living environment, age, income and so on. Based on previous experience on which customers respond to prospects the company sends out, a learning algorithm could use a relatively small set of customers to learn a mapping that classifies customers into *responsive* and *non-responsive*. After learning, the mapping could be applied to all customers to *predict* whether it would make sense to send out brochures to a particular customer, thereby maximizing the efficiency of the marketing efforts.

When the class labels are discrete symbols, as in our example, then this type of learning is called *classification*. If the mapping is required to predict real numbers, it is called *regression*. Classification could be used to learn behaviors, though the problem is that one would need correct labels (i.e. actions) for all examples (i.e. states), generating problems similar to the programming setting described above. However, we will see that supervised learning algorithms are used in the process of learning behaviors, though embedded in the reinforcement learning paradigm.

Unsupervised learning is characterized by a complete lack of feedback. Usually the goal of learning is to find a *clustering* of the problem instances. For example, a company can try to find groups of customers that are ‘similar’, in some way. Often there is some feedback that is measured in terms of how useful the clustering is for another task. Another application is to find *association rules* that express regularities in customer’s data; for example, people who buy chips often also buy beer. Unsupervised learning algorithms are often used in behavior learning systems to cluster the state space into *regions* that are in some way similar, often called *state quantization*. For example, one can cluster states that require the same action, or that have a similar distance to the goal state.

The setting that is most relevant for learning behaviors is the *reinforcement learning setting*. It is characterized by *limited* and often *delayed* feedback. Because it is the main

³Learning versus (genetically) prewiring refers to the learning versus programming dichotomy. Yet, *artificial evolution* has been used for decades as a *population-based* alternative to ML approaches (e.g. see the work by Holland, 1975). Such evolutionary approaches evolve complete populations of individuals from which the best functioning (i.e. the *fittest*) individual for some particular environment is selected.

topic of this book, we describe it in somewhat more detail in the following paragraph.

1.1.3 The Reinforcement Learning Paradigm

Reinforcement Learning (RL) (Kaelbling *et al.*, 1996; Sutton and Barto, 1998) is a learning paradigm that is – if we look at the amount of feedback that is given to the learner – positioned somewhere between supervised and unsupervised learning. In a typical RL task, an **environment** consists of a set of distinct **states**, one of which is the **current state**. In each of these states an **agent** can choose an **action** from a predefined set of actions. After **performing an action** the current state is changed to another state, based on a **probabilistic transition function**. In addition, the agent receives a **numerical reward** that is determined by a **reward function**. The objective of the agent now is to choose its actions in such a way that the sum of the rewards obtained by making transitions from state to state, is maximized⁴

The states, actions, transition function and reward function together make up a **Markov decision process** (MDP). An important aspect of MDPs is the so-called **Markov assumption**, which states that *the current state provides enough information to make an optimal decision*. That is, the agent can choose its best action by looking only at the current state; no other information is needed. For example, this is true for CHESS, but not for poker. A variety of problems can be modeled using MDPs. A **goal-based** task is one where there are one or more **goal states**. In this type, the agent only receives a positive reward for reaching such a state; on all other transitions it gets zero reward. An example of such environments is a *maze* in which the task is to find the exit. In other types of environments there is no goal state and the task simply is to maximize the total reward in the long run.

The action choices of the agent are kept in a **policy** that stores for each state the action that the agent will choose. An **optimal policy** is that policy that will gain the most reward when applied in the environment, i.e. the MDP. Now we could *program* the optimal policy directly into the agent, or we could use *reasoning*, but here we want to *learn* them. Whereas there exist algorithms that learn policies directly, most methods employ **value functions** to facilitate learning. The value function of a policy expresses for each state how much reward will be obtained in the future if we start in that state and use the policy to select all future actions. An **action value function** expresses for each state the expected reward in the future if that action is taken. An **optimal value function** is the value function of the optimal policy. If we would have the optimal value function, optimal action selection would be easy; we simply take the action that will lead us to the state with the highest (expected⁵) value. Thus, learning an optimal policy can be achieved by learning an optimal value function and to do this there are basically two types of algorithms.

The first solution algorithm is **dynamic programming** (DP). A crucial assumption is that one has complete knowledge of both the transition and the reward function. DP algorithms typically start with a default value for each state, e.g. zero. Then, they iteratively recompute the value of each state as an expected value over all transitions (and rewards)

⁴Usually, in AI, RL approaches try to *maximize* the rewards. However, in the context of *operations research* one often sees the opposite, where rewards reflect *costs* and the agent must try to *minimize* the sum of rewards (e.g. see Bertsekas and Tsitsiklis, 1996).

⁵Note that an MDP behaves probabilistically. Thus, actions can always have less-than-optimal effects, and we can only choose to maximize the *expected* value of the next state.

to other states and their values. Because all the information about the environment is known, DP algorithms can be shown to compute optimal value functions, and thus optimal policies. Note that, although value functions are iteratively improved, DP is more similar to *planning* than to learning.

The second type of algorithms is generally referred to as RL. Here, the agent has no knowledge about transition probabilities or rewards. Initially, the agent starts with a random value function and a random policy, in some state s . It chooses some action using its policy and sees the result, i.e. a new state s' and a reward. Now, if the reward plus the value of the new state is higher than predicted by the value function, the agent increases that value by a small amount. If it is lower, then it decreases the original value. In this way, the value function becomes an improved version of the original one, caused by real **experience**. And it makes sense. For example, let us assume I can normally predict that it takes me 20 minutes to get home on my bike. On some day, I start at 16:00h, and after five minutes of traveling I meet a colleague on the street and we spend 10 minutes talking. After the conversation, at 16:15h, I update my original time of arrival of 16:20h to 16:30h, because now it takes me still $20 - 5 = 15$ minutes on the bike. So, I have updated my original **prediction** of 20 minutes to 30, based on actual experience.

RL approaches learn from experience to estimate value functions. Now there are two aspects that make RL difficult. One is the problem of **delayed rewards**. For example, when playing a game such as CHESS, all rewards obtained during the game will be zero, except when entering one of the goal states, e.g. when winning the game. Depending on the number of actions taken in order to reach the goal state, learning the value of the initial state may take many games before the goal state reward is propagated to this initial state. A second challenge in RL is something that is called the **exploration–exploitation problem**. If the agent would always choose the best action based on its current value function (exploitation), it would never find out whether there are possibly better actions. So, in order to find those, it sometimes has to 'try out' worse actions that enable the agent to find other courses of actions (exploration) that might deliver more reward. Balancing this trade-off is vital for finding an optimal policy.

1.2. You Can Only Learn What You Can Represent

Talking about generic states and actions, like we have done when explaining RL, is useful to convey the conceptual ideas. Yet, when we want to build artificially intelligent systems that can learn from experience, we have to make these things explicit, and talk about the *representation* of the world (see Markman, 1999; Sowa, 1999; Brachman and Levesque, 2004, for overviews). Humans are limited by the things they can perceive using their ears, eyes, touch sensors (e.g. hands, skin), nose and mouth, which, in various forms, represent the world to them. Some things of the world are beyond our perception, such as high frequency sounds, and radio waves. Inside our heads we can form additional representations of complex concepts such as *chairs*, *government buildings*, *trust* and *time*. These representations may be built directly on top of our sensors, and additionally in terms of each other. Much is known and unknown about *cognitive* representations in humans (e.g. see Margolis, 1999; Claplin, 2002, for some pointers).

A general definition of what representation is and that applies to both human and artificial systems contains at least three elements (Markman, 1999). The **represented world**

is the domain that the representations are about. The represented world may be the world outside the (cognitive) system or some other set of representations inside the system. That is, one set of representations can be about another set of representations. The **representing world** is the domain that contains the representations. The set of **representing rules** relates the representing world to the represented world through a set of rules that map elements of the represented world to elements in the representing world. Rules induce isomorphisms when every element in the represented world is represented by a unique element in the representing world, otherwise it is called a homomorphism.

For humans and artificial systems, the lowest level of representation consists of what they perceive through their sensors. This marks a boundary between the intelligent system and the outside world, and puts a limit on what things the agent considers to be part of the real world:

Perception is Reality

Representations can be very complex, or very simple. In Figure 1.2 two *Braitenberg vehicles* (see Braitenberg, 1984, for many interesting vehicles) are depicted. The only level of representation that is present consists of two sensors that detect light. In the left vehicle the right sensor is connected to the right motor and this will make the vehicle back away from the light in the current situation. In the right vehicle the left sensor is connected to the right motor (and vice versa), which makes this vehicle to move towards the light source. Both vehicles do not introduce any more sophisticated level of representation, but still they perform a simple behavior⁶ consistently.

This shows how powerful a couple of such simple control structures and representations can be. In contrast, many other types of architectures for intelligent behavior are like the one in Figure 1.3. In this *cognitive architecture* (see Langley, 2006, for more examples) the control mechanism has a much more complex structure, both in terms of the representations that are used (e.g. the agent's beliefs about the world, descriptions of goals it must achieve, and predefined plans to achieve sub-tasks), and in terms of the *algorithmic* structures that are needed to decide on an action based on all the constituents of the agent's mind. In Chapter 7 we go into more detail on this kind of architectures, and more specifically we focus on how learning can be incorporated.

Before discussing which types of representations are used in AI systems, we may first raise a question on *how much* representation we need and how they come about. This has been the subject of many debates in the past decades. The problem of how representations relate to the real world is essentially the *symbol grounding* problem (Harnad, 1990), but we ignore it and only consider the situation where there is *some* representation of the world to

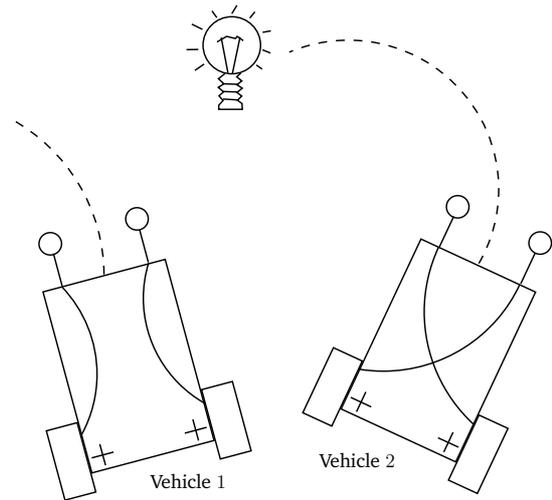


Figure 1.2: *Braitenberg vehicles*.

⁶Pfeifer and Scheier (1999, pp. 475–478) describe interesting experiments using a group of such robots that seem to "tidy up" a room with obstacles.

begin with. The other issue has been subject of debate during the eighties. Brooks (1991) provided the start of developments in *behavior-based* architectures (Arkin, 1998) such as the *subsumption architecture*, by proposing that intelligent behavior could be achieved by a large number of loosely coupled processes that mostly function in an asynchronous and parallel way. He argued that internal processing would have to be minimal and that sensory signals should be mapped relatively directly to motor signals, as in Braitenberg vehicles. In essence, this called for *less* representation and abstraction. Later, this grew into the field of *embodied intelligence* (Pfeifer and Scheier, 1999, or, *new AI*), which emphasizes the fact that behavior consists of a *bodily activity* in a physical world and that we must understand intelligence in terms of the interaction between the embodied system and the environment.

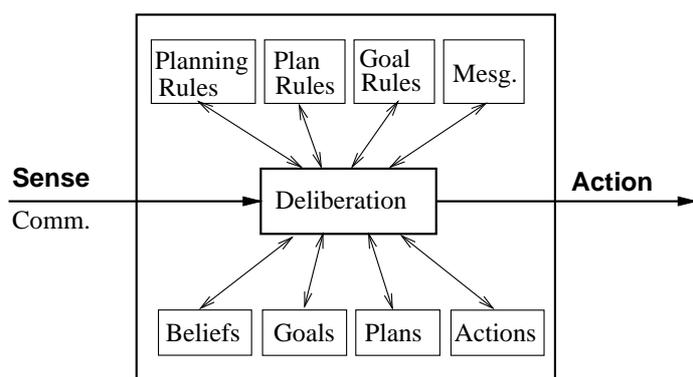


Figure 1.3: A cognitive agent structure.

A central motto⁷ is: *"The world is there, no need to remember it all"*. This contrasts with what is often referred to as *the computer metaphor* of seeing intelligence as information processing or the manipulation of abstract symbols. Other approaches in learning and evolving behaviors show the potential of such reactive approaches (e.g. see Nolfi and Floreano, 2000; Nolfi, 2002), but in this book we argue that

representation is important to be able to scale up to larger problems and to insert and extract knowledge from the learner (see also Markman and Dietrich, 2000b).

1.2.1 Generalization, Abstraction and Representation Formation

The most important aspect of a learning process is *generalization*, which is the capability to use information learned from one situation in other situations that are in some way "similar". In daily life, we do it all the time. For example, when going to a conference in a country to which we have never been before, we often experience little to no problems when using public transportation at that location. We can do this because the process of using them is quite similar in many countries: you have to look at the departure schedules, find out which line you require, buy a ticket, get into the right vehicle and get out at your destination. It does not matter much that the trains have different colors, or that stations may be built and structured in various ways. However, for generalization to work, we must have some idea of whether a new situation is sufficiently similar to already experienced situations and that we transfer the right aspects of this experience.

"We're always learning from experience by seeing some examples and then applying them to situations that we've never seen before. A single frightening growl or bark may lead a baby to fear all dogs of similar size – or, even animals of

⁷This is similar to the fairy tale of *Hop o' My Thumb* who dropped bread crumbs to find his way back. In this way, he would not have to remember everything about how to get back; he only needed to *modify* its environment and simply follow the trail of bread crumbs.

every kind. How do we make generalizations from fragmentary bits of evidence? A dog of mine was once hit by a car, and it never went down the same street again – but it never stopped chasing cars on other streets.”
(Minsky, 1985, Section 19.8)

A generic generalization process requires a representation space and a *similarity measure* that defines for each pair of representations how similar they are. A similarity measure induces a *distance* in the representation space. Now representations can be grouped according to the measure and generalization takes place among situations that are near in that space. This makes generalization completely dependent on the representation space.

Similarity is Proximity in Representation Space

More complex representations offer more opportunities to construct such similarity measures, but at the same time they introduce more choices that have to be considered.

A broader view upon generalization is that it introduces a form of *dynamic representation*, or *representation formation*. As already said, representations can be *about other representations*, and generalization can be seen as building higher levels of *abstraction* in a new representation space (see also Korf, 1980). For example, based on the representations of a mouse, a keyboard, a monitor and a system, one can build the higher-order concept of *computer*. Building more complex representations from simpler ones is generally called *constructivism*, and has its origins in the *cognitive development* theories in psychology (Piaget, 1950; Thornton, 2002). In AI, constructivist approaches are often based on neural networks (Elman *et al.*, 1996; Quartz and Sejnowski, 1997; Westermann, 2000), but many other types have been described (e.g. see Drescher, 1991; Thornton, 2000).

AI has introduced many types of representations, including *sub-symbolic* ones as used in neural network approaches and purely symbolic representations such as propositional logic. Popular representation schemes include Bayesian networks, relational databases, rules, trees, graphs and many more. In the end, general AI systems should employ a whole range of different representations depending on their suitability in various sub-tasks in the intelligent architecture (see also Minsky, 1985). In this book we are mainly interested in a division in three fundamental *classes* of representation that have to do with how the intelligent system perceives the world. These are *atomic*, in which the environment's state is perceived as a single symbol, *propositional*, in which the world is structured in terms of propositions, and *first-order*, in which the current situation is perceived as consisting of *objects*. In the following we illustrate these classes using three imaginary robots.

1.2.2 CANTOR: Representing the World in Snapshots

Our first robot, named CANTOR⁸, is very simple. Each possible state is represented as an atomic *thing*, e.g. a *symbol*. For ease of explanation, we assume that CANTOR stores its value function and policy in a small notebook. On each page, it stores a state with an action value table for that state, see Figure 1.4a. Here we also see that CANTOR has experienced several learning steps in which it has once decreased the value for action a and increased it twice for action b.

⁸This robot can only reason in terms of sets. It cannot generalize using the structure of the elements in these sets. Georg Cantor is well-known for his contributions to *set theory* (see Davis, 2000).

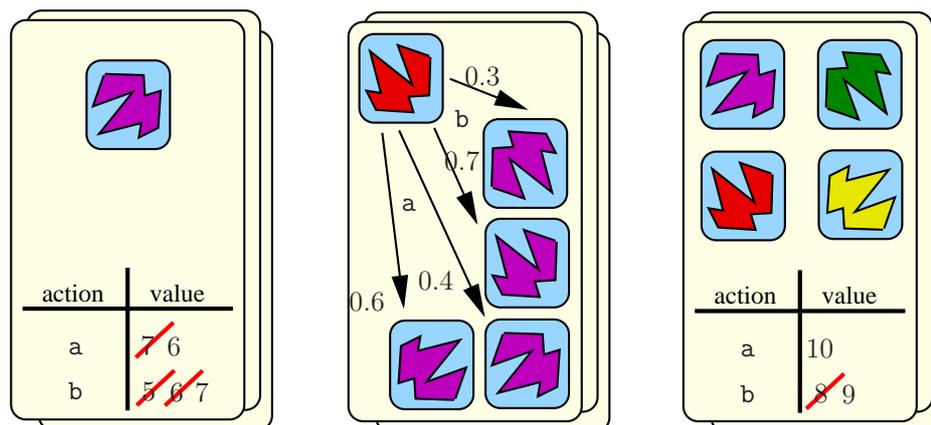


Figure 1.4: Data structures for CANTOR. a) Part of the state-action value function (Q). b) Part of the transition model (T). c) Part of a state-action value function with state aggregation.

At each step in the world, CANTOR observes the current state and looks it up in its notebook. Based on the values in the figure it would choose action b in this state in case it would not explore. Depending on the total number of states, looking up the current state may be a time-consuming operation. Stored in a computer memory, this may not seem too problematic. However, let us assume the states are photos, taken by a camera mounted on the robot. In a physical world, the number of distinct photos of the robot's surroundings is enormous. Every two photos that differ only slightly in a single pixel are completely different for the robot, and they get a different page in CANTOR's notebook.

The same storage and retrieval problems occur when a model is available. For each of the states, CANTOR would have to keep a page such as in Figure 1.4b, in which the non-zero transition probabilities to all other states must be stored. Depending on the stochasticity in the environment, each page might end up storing all states.

Generalization, Abstraction and Representation Formation. The possibilities for generalization for CANTOR are very limited. Even though there could be many states that are almost identical, as can be the case with photos, CANTOR can only see whether two states are exactly identical or not. What CANTOR can do to generalize is to *group* of states once experience shows that they have similar action values. CANTOR can then replace all pages of the states in one group by just one page that contains all states in that group and one action value table, see Figure 1.4c. In this way, states *share* information about action values and each time CANTOR visits a state in that group, implicitly all action values of all the group's states are updated at once, thereby generalizing over that set of states. Note that this implies that from the moment of grouping, all states will have equal action values, and it depends on whether the grouping is 'right' how this will affect future experiences and with that, the possibility of learning an optimal behavior.

1.2.3 BOOLE: Representing the World in Twenty Questions

Although CANTOR can – in principle – learn optimal behaviors, it is not of much use for most applications. Therefore, let us introduce the more advanced robot BOOLE. For this robot, state information is *decomposed* into a small number of indicators. Each such indicator represents the presence of a relevant aspect of the state. For example, there could be an indicator for the presence of a wall in front of the robot, or it could indicate whether

Figure 1.5: Data structures for BOOLE. **a)** Part of the state-action value function (Q). **b)** Part of the transition model (T), where a_i stands for the answer to question q_i . **c)** Part of the state-action value function with state generalization. **d)** A state value function with linear function approximation.

or not the robot is carrying a load. The general form of such a state representation can be seen as a list of *answers to questions*⁹. The questions are fixed, and are part of the robot (e.g. its sensors). Each answer can be either *boolean*, i.e. true or false, or *real-valued*. The technical term for such representations is *propositional*¹⁰, or *feature-based*, and it is the most common representation in many AI or ML systems.

Now, each page of BOOLE’s notebook contains for each state a distinct set of answers to the questions in the feature set, see Figure 1.5a. BOOLE’s learning process is similar to that of CANTOR. First it gets a list of answers, which it looks up in its notebook. Then, based on the action values for that state, the robot chooses an action and perceives the next state, i.e. set of answers, and a reward.

Storage and retrieval of states can be made easier by looking at the *structure* of states. For example, BOOLE can have a separate part in its notebook for all states in which the first question is answered *yes*. And then another division in these parts based on the answer to the second question and so on. In this way, looking up a state can be done more quickly than CANTOR did¹¹. More importantly, BOOLE can *emulate* CANTOR’s representation by introducing one question for each state in CANTOR’s representation. Such a question only asks for “is it this particular state?”. Thus, each state in BOOLE’s representation would consist of a list of all *no*’s except for one *yes*. In other words, BOOLE can do everything CANTOR can, but not the other way around.

Generalization, Abstraction and Representation Formation. BOOLE’s representation of states offers many opportunities for generalization and abstraction. For example, the specification of a transition model can make use of abstraction over effects of actions on separate features of the state. For example, Figure 1.5b shows a part of such a model. Here, it specifies that in all states where the answer to the first question is *yes* and either

⁹The nature of this representation is similar in spirit to the game “who is it?”. In this game one has to find out who – out of a group of dozens of individuals – one’s opponent has in mind. By asking questions such as “is this person female?” or “does the person wear glasses?”, one has to guess the person’s name in as few questions as possible.

¹⁰Propositional logic is also known as *Boolean logic*, named after George Boole (see Davis, 2000, for a description).

¹¹We have deliberately used pictures for CANTOR to highlight the concept. However, if CANTOR’s state representation would consist of numbers, for example, it could use more efficient means such as hash tables.

the second answers no or the fifth answer is larger than 3.0, the action a can have two possible effects. In one – with probability 0.8 – we know that (and how) the answers to the first and fifth question are changed, with respect to the current state in which the action was performed, and in the other – with probability 0.2 – we can see the effects on the answers to questions three and five. Such abstractions presuppose the notion of a propositional *language* to describe the model in this way. Note that where CANTOR needed such a description for each state separately, BOOLE describes transitions for a whole set of states simultaneously.

A similar type of generalization can be used to store the value functions. Instead of storing such values for each state separately, as CANTOR did, BOOLE can group states that share the same answers to selected questions. Figure 1.5c shows a state-action value function in which all states that share the same answer to question three, and additionally have an answer to the fifth question that is lower or equal to 10.0, are grouped. Depending on the number of different answers to the fifth question, such generalization may group many states. Another advantage of BOOLE’s representation is that it can generalize over states *it has not seen yet*. For example, the page in Figure 1.5c generalizes over the state $\langle \text{yes, yes, no, no, 2.9} \rangle$, such that CANTOR may know which action is better in that state even though it may not have been there yet. Obviously, such state generalization is only possible if it is known from experience that states can be grouped.

One of the main advantages of BOOLE’s representation is that states can be interpreted as *vectors* in an n -dimensional space, where n is the total number of questions. This n -dimensional space offers many possibilities for generalization, and the *similarity* of states can be equated with the *Euclidean distance* in that space. For example, if the current state has not been visited before, but there is a nearby state (in this space) in BOOLE’s notebook for which the optimal action value is known, then it may carry over this knowledge to this new state. Similarly, the value of a state can be expressed as a linear weighting of the feature values, which is depicted in Figure 1.5d. In this way, learning a value function comes down to tuning only five *weight parameters*, independently of how large the complete set of states is. This use of generalization forms the basis for many types of neural networks and other approximation architectures that are used in RL.

1.2.4 FREGE: Representing the World in Terms of Objects and Relations

Propositional representation is useful for modeling various tasks. However, it lacks the expressive power to capture a number of important general forms of reasoning, in particular *reasoning about individuals, the properties they possess and relations between individuals*. Furthermore, such representations cannot *quantify* over individuals, i.e. say that some property holds *for some individuals*, or *for all individuals*. For this, one needs *first-order logic*. Many dialects exist, though the common factor is that they represent and reason over structures containing *objects* (i.e. individuals) and *relations* between them.

Representing the world in terms of objects is most natural from a number of different perspectives. For humans it seems the most intuitive way to structure the world, (see Dennett, 1987, on *the intentional stance*). Objects often tend to refer to *coherent entities of physical material* though we have no problems with mental objects such as *pride, love* or *the square root of -4.33* . One possible explanation is that evolution has endowed us with these capabilities because they can be very efficient. *“The description of the world in terms of objects and simple interactions is an enormously compressed description.”* (Baum, 2004, p.

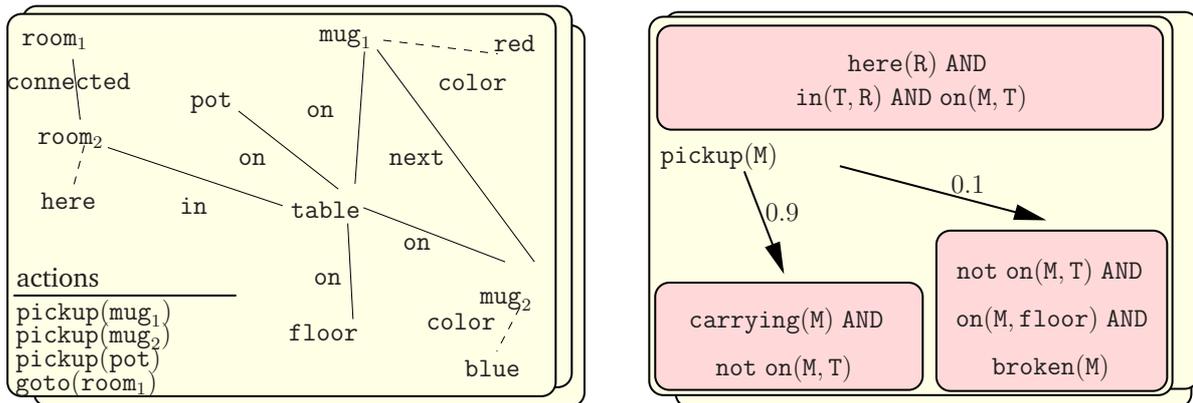


Figure 1.6: Data structures for FREGE. a) A state and a set of applicable actions. b) Part of the transition model (T).

168). In AI systems, the power of relational representations has been recognized since the beginning. Mainly because of reasons concerning computational complexity, using these representations for challenging tasks such as RL and other types of environments where there is a significant amount of uncertainty, has begun fairly recently. But in the end, relational representations are vital for scaling up to more complex problems. *“It is hard to imagine a truly intelligent agent that does not conceive of the world in terms of objects and their properties and relations to other objects”* (Kaelbling et al., 2001). Fortunately, for many types of computer-based tasks, relational representations are wide-spread, for example in relational databases.

Let us take a look at our third robot, FREGE¹². From all three robots, FREGE uses the richest representation. An example is depicted in Figure 1.6a. A number of the keywords in the graphical representations are objects. These are `table`, `mug1`, `mug2`, `room1`, `pot`, `floor` and `room2`. Solid lines between objects denote relations. For example, the objects `table` and `floor` are connected by the relation `on`, meaning that the table is on the floor. Dashed lines denote *attributes*, or unary relations. These types of relations denote certain *properties* of an object. For example, we can see that the color of `mug1` is `red`. An interesting aspect of such a relational representation is that actions are now *parameterized* with objects. For example, the action `pickup(mug1)` denotes picking up the object `mug1`. This causes the number of actions to be dependent on the number and type of objects in the current situation. If there would be just one more mug `mug3` on the table, presumably an additional action `pickup(mug3)` would be applicable. This is another advantage of relational representations; in contrast to BOOLE’s fixed-length question list, FREGE’s representation can vary in size with each state.

A relational representation provides much detailed information about the state. A natural question to ask is whether the same world could be described by BOOLE. In principle, the answer is positive, but it comes with some problems. For each possible relation (including unary ones) we have to create a question. For example, if the colors include `red`, `blue`, `yellow` and `green`, BOOLE would need – for each object in the system – four questions such as *“is the color of the first mug red?”*, *“is the color of the first mug blue?”* and so on, and only one of these four questions would be answered with `yes`. This becomes

¹²Gottlob Frege (Frege, 1879) is the founder of modern approaches in first-order logic (see Davis, 2000, for a historical overview).

more troublesome with more objects and relations between more objects. An additional problem is that the questions (and their order) completely depend on the exact number of objects. FREGE's representation, on the contrary, *scales* easily with the number of objects.

Generalization and Abstraction and Representation Formation. Because there is much information present in FREGE's state and action representation, there are many possibilities for generalization and abstraction. The most common way is to introduce a *formal language* that can express certain properties of the states and that lets FREGE *reason* about them. For example, a logical formula such as $\forall M \text{ on}(M, \text{table}) \rightarrow \text{blue}(M)$ expresses the phrase *all things on the table are blue*. This is a false statement because there is at least one red mug on the table too. The M symbol is a *variable* that can stand for any object that satisfies some property, in this case being on the table. The \forall symbol is a *quantifier* that enables to express properties that must hold for all objects in the current state, and is independent of how many or which types of objects are present. This shows that FREGE's representation is a lot more general than that of BOOLE, that uses a fixed set of questions.

Figure 1.6b employs a very simple language to specify the effects of the `pickup` action. It says that if the robot is in a room R , and there is some object T (which presumably is the table here), and the robot tries to pickup the object M that is on T , then two things can happen. With 0.9 probability, the robot's intended action succeeds and it is carrying the object and it is not on T anymore. Yet, with a small 0.1 probability, the action fails and the object M is not on T anymore, but it has fallen onto the floor and it is broken¹³.

Formal languages such as used in the examples are very powerful in expressing generalized statements about the world, yet automatically generating such descriptions is computationally hard, especially when combined with probability and utility. BOOLE's representation offers a simple similarity measure for generalization if one interprets the answers to the questions as a vector in an n -dimensional space. Because of the symbolic nature of FREGE's representation, this trick no longer exists. Nevertheless, relational representations offer many opportunities to generalize over objects through the use of variables combined with quantifiers, when learning value functions and policies. And in the same way actions can be parameterized, so can goals. For example, when learning how to deliver a specific mug to a specific room, the robot can generalize its policy (by using variables) to be able to deliver *any* mug to *any* room. Another example is that a policy that is learned for stacking 5 objects can be used for any number of objects, which is something that cannot be achieved by BOOLE.

1.2.5 The World Might be Larger than We See

So far, we have assumed that everything that might be important for a current, optimal action, can actually be perceived. Obviously, this will not be the case for most general environments, and certainly not for our own, human, real world. There are many things we cannot perceive with certainty, or maybe not at all. Think of a some card game you play with multiple people. You first have to know the rules of the games, and you might have some general strategies but your knowledge about the current state is restricted to

¹³Interestingly, a full specification of the failing action would in general be impossible. This is because one would have to specify all the different ways in which it can be broken, which pieces are created in breaking the mug, and where they are. A full symbolic specification of all these possibilities combined with a proper probability distribution over all these possibilities is impossible, though Zettlemoyer *et al.* (2005) describe a useful trick to handle such outcomes as *noise* in the probability distribution.

what you can see, which are your own cards. Furthermore, there is also uncertainty about what other people will do with their cards, and which cards are still in the deck.

The *partially observable* MDP (POMDP) model extends the MDP. In a POMDP the agent does not perceive the actual state, but only gets an observation that probabilistically depends on this state. One general solution in this type of environments is to represent the agent's current state by a probability distribution over all states that gets conditioned on the observations. Many types of abstraction and generalization can be employed in this setting too. POMDP solution techniques are more complex, and exact solutions can only be obtained for modest-size problems. Many recent techniques for POMDPs focus on approximate solutions. This book is about fully-observable MDPs because virtually all current first-order methods are restricted by the Markov assumption, and it is here that we draw a boundary. Still, when employing abstraction and generalization in MDPs, some of the same problems of non-Markovian environments arise. That is why, in Section 2.7, we briefly describe what happens when one goes beyond the Markov assumption and we discuss some intuitions behind POMDPs and *predictive state representations*.

1.3. About the Contents and Structure of this Book

Basically, we are interested in representations and algorithms that can be used to build a robot such as FREGE. But in order to do just that, we need to take a close look at techniques that were used to construct the previous two robots CANTOR and BOOLE. Presumably, a lot of experience and insights have been gained by trying to build these two robots. FREGE will operate in a much more complex, and much more structured world, but it will also share concepts with CANTOR and BOOLE such as learning evaluation functions, predicting what consequences actions will have and how to generalize experience.

Investigating how to build intelligent robots such as FREGE is our main goal, but at the same time, it is part of the current focus of development in at least three distinct sub-fields of AI. In this section we first describe the main motivations and theme of this book at the crossing of these three distinct research fields. After that we outline the book's structure and give a short preview of the contents and structure of its chapters.

1.3.1 Main Theme of This Book

This book is about modeling and computing behavior, with a special emphasis on *representational* aspects. One of the reasons for pursuing this line of research was that a couple of years ago there were almost no techniques for utility-based learning in first-order domains. Yet, it is desirable because of technical reasons, e.g. scaling up to larger – and more complex – problems, but also because for possible connections to other contexts, such as natural language, humans, and general cognitive agent architectures. Conceptually, RL seems to be the most natural candidate as a learning technique for agents. Most state-of-the-art formalisms that are used for programming intelligent agents are based on first-order logic, but almost all of the work in RL was occupied with efficient means for generalization in propositional domains. For example, the use of neural networks was widespread, but such techniques cannot handle rich representations consisting of arbitrarily-sized domains of objects and various relations between these objects. Furthermore, the use of (symbolic) domain knowledge is usually not considered in RL, though it has been shown very useful in supervised learning approaches in first-order settings. Coming up with general solutions

requires one to look at a whole range of first-order representation, learning and generalization techniques. Therefore, the main purpose of this book is to find answers to the following research questions.

How can MDPs be posed over first-order domains, consisting of objects and relations, how can these be solved using either model-based DP techniques, or simulation-based methods such as RL, what are the characteristics and challenges of this setting, and how is it related to existing, propositional techniques?

In this book we emphasize the representational aspects of the first-order setting. We focus on an agent that is put in an environment that consists of objects and relations between objects, and its goal is to learn a policy for a sequential decision making problem that is guided by reward feedback. Obviously, new representations require new algorithms in general, but as we will see, most of the algorithmic knowledge and achievements in utility-based learning that already exist for propositional representations, can and will be reused. Our personal insight into the incapability of techniques such as neural networks to cope with RL in first-order domains, is paralleled by natural developments in three distinct – but related – research fields. These fields are RL, first-order ML, and (agent) planning languages, and we describe them briefly below. Doing so gives us an opportunity to place the material in this book in its proper historical context. It also shows how learning sequential decision making in first-order domains fits into the natural developments and maturation of these fields.

1.3.1.1 REINFORCEMENT LEARNING

The first, and main, field that is extended by the material in this book is generally referred to as RL. As do some general books in this area, such as those by Sutton and Barto (1998), Bertsekas and Tsitsiklis (1996) and Si *et al.* (2004), it includes sample-based, model-based and various approximate techniques, all centered around MDPs and its extensions. Sutton and Barto (1998, pp 16–23) describe at length three main threads that run up to the mid nineties, which are **1)** trial-and-error learning started in the psychology of animal learning, **2)** optimal control learning and solutions using value functions and DP, and **3)** temporal-difference methods. These threads came together at the end of the eighties to form the modern field of RL. Treatments of the history of RL can be found in many texts, for example in Sutton and Barto (1998)’s book, in general AI books such as Russell and Norvig (2003)’s, and in other overviews, for example (Kaelbling *et al.*, 1996; Keerthi and Ravindran, 1997; Barto and Dietterich, 2004).

Much about the model-based setting for MDPs was known before that starting from early work on efficient solution concepts and algorithms such as the Bellman equations and value iteration (Bellman, 1957) and policy iteration (Howard, 1960) and the connection with shortest path algorithms such as Dijkstra’s algorithm. Many of these approaches are described in the standard textbook by Puterman (1994). Early work that includes ideas of the model-free setting such as trial-and-error learning are Samuel (1959)’s CHECKERS playing program, Holland (1975, 1986)’s work on evolutionary adaptive algorithms and the bucket brigade algorithm, but some ideas can be traced back to early work by Minsky and even Turing in the early fifties (see Sutton and Barto, 1998, Section 1.6 for an excellent overview). The seminal paper by Sutton (1988) marks the beginning of the modern field

of RL with which we start in this book. In this paper, Sutton separated learning value functions from learning control, introducing the general concept of *prediction*.

In the early nineties, many developments can be found in employing *value function approximation* methods, for example many using neural networks (Lin, 1992) and some using decision trees (Chapman and Kaelbling, 1991). In this period algorithms and proofs for Q -learning (Watkins, 1989; Watkins and Dayan, 1992), SARSA (Rummery and Niranjan, 1994; Rummery, 1995) and other one-step algorithms (Singh *et al.*, 1995) were developed. Much of the research was focused on applying RL algorithms and using various approximation architectures to learn generalized value functions. During the nineties there were many developments for the model-based setting too (thoroughly surveyed in Boutilier *et al.*, 1999) in the form of *decision-theoretic planning*. Structured abstraction schemes for states, and algorithms that make use of this structure during computation, have been developed in the context of *factored* representations of MDPs, value functions and policies by, for example, Dearden (2000), Boutilier *et al.* (2000a) and Dean and Givan (1997). Other developments during the nineties focused on learning policies directly, in contrast with value-based methods. Early approaches include the REINFORCE algorithm (Williams, 1988, 1992), and later other approaches appeared (e.g. Sutton *et al.*, 2000). A large class of other related methods that deviate from online value-based methods are *evolutionary approaches* (Moriarty *et al.*, 1999), which use evolution to find policies. During the nineties, and especially from the second half of it to now, much progress has been reported on yet another type of models and algorithms, specifically designed for *temporal abstraction* and *hierarchical decomposition* of policies. Identifying sequences of actions as *behaviors*, the field of hierarchical RL has found new ways of modularization of the learning process (see Dietterich, 2000b; Barto and Mahadevan, 2003; Ryan, 2004a, for overviews). Early work by Kaelbling (1993a), Dayan and Hinton (1993) and Thrun and Schwartz (1995) was quickly followed by widely used approaches such as HAMQ (Parr and Russell, 1998), MAXQ (Dietterich, 1998) and OPTIONS (Sutton *et al.*, 1999).

An alternative view on historical developments in RL was sketched by Sutton (1999), and it consists of three periods. The past encompasses the period roughly until 1985 in which the idea of *trial-and-error* learning was developed, and includes all approaches back to the early fifties. The present (in 1999) was about value functions that were formalized, and the construction of value function approximation schemes, and proofs of convergence. The future or RL (again, in 1999) is about *constructivism*, i.e. to *take a further step away to focus on the structures that enable value function estimation*. In this view, we are far into this period, and indeed many approaches that have been developed in the recent decade are about this constructivism. Not only have there been many new types of representation and approximation schemes developed, but also algorithms that *develop their own representations*, which is particularly useful for the general goal of utility-based learning in unknown environments. For example, learning structured models of MDPs (e.g. see Littman *et al.*, 2005; Degris *et al.*, 2006; Jonsson, 2006) is a constructivist extension of the work on factored MDPs during the nineties. Early on, for hierarchical approaches, this type of constructivism was investigated in, for example, HQ (Wiering and Schmidhuber, 1997) and HEXQ (Hengst, 2002). In the past decade, many such algorithms have been developed as extensions to MAXQ, HAMQ and OPTIONS. The automatic construction of *basis functions* to describe the world is a topic of much research lately in DP algorithms (e.g. see Keller *et al.*, 2006) but also in more generic contexts in which general characteristics

of the environment are extracted (Mahadevan and Maggioni, 2007),

Much about all these developments in MDP algorithms and the employment of abstraction and generalization can be found in Chapters 2 and 3 in this book. We describe the main types of algorithms and construct a general classification based on the type of generalization (e.g. of value functions) and whether representations are constructed by the learning system. Much recent RL research has focused on more general contexts such as *partially observable* MDPs where there is uncertainty about the true state, though most of this is beyond the scope of this book (but see Section 2.7).

So far, we have described some important historical lines that are mainly concerned with the *algorithmic* side of RL approaches. If we focus on the *representational* dimension, we obtain another picture of the field, one that is most relevant for this book. The early work in MDP research, both model-based and model-free approaches, used *atomic* state representations, which is the CANTOR setting. In this setting, states have no inherent structure and most results are about the algorithmic aspect: how to learn state values and optimal policies. The second period, mostly concentrated around the nineties, is about *propositional*, or *feature-based*, representations of states, the BOOLE setting. Many representational devices can handle these types of state representations for abstraction and generalization purposes, such as neural networks, propositional rules, decision trees, support-vector machines and many more. In this setting, states have propositional structure and generalization can make use of that. Most RL approaches use some kind of generalization that is tailored to these representations, and at the end of the nineties propositional representations were the state-of-the-art in RL.

Yet, around the turn of the century, approaches started to appear that extended the state, and action, representations to the *first-order* case in which the world is described in terms of *objects* and *relations*; the FREGE setting. The first was an application of first-order decision trees as a value function generalization technique for Q -learning in a small BLOCKS WORLD domain by Džeroski *et al.* (1998). They were inspired by, as is noted in that paper, the invited talks (at IJCAI-97) by Richard Sutton and Leslie Kaelbling who both suggested the combination of first-order learning algorithms and RL. Soon followed the first DP technique for first-order representations by Boutilier *et al.* (2001) who defined a value iteration algorithm that operates over a first-order logical MDP specification in the situation calculus. These two approaches initiated the third, representational period in MDP research, which is the main topic of this book: MDPs defined over first-order representations of the world and efficient algorithms for solving them. First-order approaches extend the applicability of MDPs to new, and larger domains, and in addition they create new possibilities such as parameterized actions, learning for multiple environments simultaneously and learning in indefinite or infinitely large environments.

The three representational dimensions in MDPs currently coexist. But more importantly, as we argue in this book, the historical developments in each of these dimensions are heavily co-dependent. Propositional approaches are usually built on top of results in the atomic approaches, in the sense that when generalization and abstraction make use of the propositional state structure they must obey certain (algorithmic) principles that are defined in the atomic case such as formulated in the Bellman equations for individual states. In a similar way, the first-order dimension borrows from the preceding developments, in particular from existing propositional mechanisms for generalization and abstraction over propositional MDPs. At the end of Chapter 3 we make these connections

explicit after which we describe the first-order dimension in full in Chapters 4 to 7.

1.3.1.2 MACHINE LEARNING

Knowing that RL is a subfield of machine learning (ML) we might expect somewhat similar developments. However, general ML has used first-order logical representations for several decades. In fact, many of the now popular approaches such as Bayesian networks were developed much later than these logical approaches. Some ML approaches can be traced back to the early days of AI (see Mitchell, 1997, for an excellent description).

First-order learning was much developed during the early seventies. One of the main developments was the formalization of learning in first-order clausal logic by Plotkin (1970). Other earlier approaches in the context of PROLOG, clausal logic and logical rules can be found in the works by e.g. Shapiro, Michalski and Winston. The study of (supervised) first-order ML, mainly in clausal logic, became a field of its own around 1990 (see Muggleton and De Raedt, 1994; Lavrac and Džeroski, 1994; Bergadano and Gunetti, 1995, for overviews). It became known as *inductive logic programming* (ILP) and was aimed at learning essentially PROLOG programs from data in a supervised setting. Much emphasis in ILP is placed upon a priori *knowledge* that can be used in the induction process. More recently, even *higher-order logical* learning approaches have been developed along the same lines (Lloyd, 2003).

Much of what is now generally understood as ML is centered around propositional representations and popular techniques such as decision trees and neural networks. Neural networks are computational structures that mimic some of the functioning of the human brain. They have been around since 1943, but after some initial setbacks it took until 1986 when they were revived by a general technique called *backpropagation* and the work by Rumelhart and McLelland. Since then they have become very popular (Bishop, 1995; Haykin, 1999; Reed and Marks II, 1999). Other popular techniques include decision trees, fuzzy logic and genetic algorithms, also commonly referred to as *computational intelligence* techniques (Jang *et al.*, 1997). Since the early nineties many *probabilistic* techniques have made their way into ML. Techniques such as *Bayesian networks* and *hidden Markov models* have become very popular for learning probability distributions from data. More recently, *support vector machines* and *kernel-based approaches* (Schölkopf and Smola, 2002) have become popular, general ML approaches. What all these approaches have in common is that they use propositional (feature-based) representations and that they can handle uncertainty, in both supervised and unsupervised settings. Another aspect is that most of these statistical approaches focus on *parameter estimation* rather than on *model selection*. Some textbooks focus entirely on the propositional setting (see e.g. Hastie *et al.*, 2001; Alpaydin, 2004) whereas others treat both the propositional and first-order setting (Langley, 1996; Mitchell, 1997; Poole *et al.*, 1998).

Coming back to first-order ML, a drawback of many of the early ILP approaches was that they could not handle noise or uncertainty well. Starting in the mid-nineties, a new wave of *probabilistic* extensions to ILP approaches appeared, under the general name of *statistical relational learning* (SRL). The foundations of this approach are based on ILP methods and a long history of approaches in combining logic with probability (Halpern, 2003; Galavotti, 2005). In the same way first-order approaches in RL can incorporate techniques from propositional RL, many popular propositional ML algorithms have been upgraded to the first-order case. Examples in the supervised learning setting include

Bayesian networks, hidden Markov models, decision trees and rules (see De Raedt and Kersting, 2003, for an overview). Unsupervised techniques involving distances, kernels and clustering techniques have made their way to the first-order case (Ramon, 2002; Gärtner, 2003), and even neural networks (Bader *et al.*, 2006) and genetic algorithms and classifier systems (Divina, 2006; Mellor, 2007). For other general overviews see (Džeroski and Lavrac, 2001b; De Raedt and Kersting, 2004; Getoor and Taskar, 2007). All of these approaches upgrade propositional ML approaches to the first-order case, but all of them are based on either a supervised or unsupervised learning setting.

Now we have two main historical lines along which one can place the material in this book. On the one hand, first-order extensions to RL extend traditional approaches that combine ILP methods and probabilistic aspects with yet another element, namely utility. On the other hand, they extend the field of SRL, that has focused so far on supervised and unsupervised settings, with an extra paradigm, namely that of RL. In both ways, first-order RL approaches extend the field of ML with a combination of first-order logic, probability, learning and utility.

1.3.1.3 ACTION LANGUAGES, PLANNING AND AGENTS

Classical planning is one of the oldest AI subjects. Starting with the early work by McCarthy (1963) on the situation calculus, AI has been occupied with modeling and axiomatizing changing world and *commonsense reasoning*. Many such logics exist and many are based on first-order logic (see Gelfond and Lifschitz, 1998; Reiter, 2001; Russell and Norvig, 2003; Brachman and Levesque, 2004; Mueller, 2006). However, it turns out to be very difficult to completely specify a world such that an automated system can compute a course of action that will reach some specified goal. One of the biggest problems is that the system can be overwhelmed by irrelevant actions. The key is to have a language that is expressive enough for interesting problems, but restrictive enough to allow efficient algorithms to operate over it. The basic representation for many classical planners is the STRIPS language (Fikes and Nilsson, 1971). It can specify some simple properties of states and how things change when actions are applied. One crucial assumption is that anything that is not specified does not change, which eliminates many irrelevant effects to be dealt with. An extension of the STRIPS formalism is the ADL language by Pednault (1989). ADL is more expressive than STRIPS because, for example, it supports *conditional* effects and quantified variables in goals. Many other variations and extensions have been introduced afterwards and later these were systematized within a standard syntax called *planning domain definition language* (PDDL) (Ghallab *et al.*, 1998). Extensions to planning such as the incorporation of *hierarchies* (as in hierarchical RL or in planning e.g. see Erol *et al.*, 1994) are also supported by PDDL.

In addition to restricted languages that are aimed at planning domains, a wide variety of action formalisms and logics exist for more general purposes. Examples include the *fluent calculus* (FC) (Thielscher, 1998), the *event calculus* (EC) (Mueller, 2006) and the *situation calculus* (SC) (Reiter, 2001). Many of these languages are constantly being extended to incorporate modal constructs, such as *belief* and *knowledge*, aspects of *time* and *duration*, *domain knowledge*, and even *emotions*. Much of the research is focused on theoretical properties of the logics and their extensions. Some have evolved into full programming languages such as GOLOG (Levesque *et al.*, 1997) based on the SC and FLUX (Thielscher, 2005) based on the FC. Finzi and Lukasiewicz (2004a) extend GOLOG with

game-theoretic constructs to deal with *multi-agent* domains. In fact, the *agent* literature (Weiss, 1999; Ferber, 1999; Wooldridge, 2002) has produced many agent programming languages that have much in common with the approaches based on FC, EC and SC.

A complete description of all approaches fills many books. What is important to note here are two things. One is that there are many action languages that are based on first-order logic. The other is that most are targeted at *deterministic* domains. Some planning approaches are able to deal with *probabilistic* action effects (Kushmerick *et al.*, 1995; Blythe, 1999) but work on this is still far more limited than on the deterministic setting. The extension towards MDPs, with reward-based goals, is known as *decision-theoretic planning* (Boutilier *et al.*, 1999). What a *plan* is for classical planning, is a *policy* (or, *universal plan*) for probabilistic environments such as MDPs. Extending first-order formalisms to deal with the *specification* of MDPs is relatively straightforward based on existing languages. Some formalizations appeared earlier than the work we have described in the RL historical setting (see for example Poole, 1997a; Geffner and Bonet, 1998) but it was not before the SDP approach (Boutilier *et al.*, 2001) that *solution algorithms* for such MDPs were developed.

Research in planning approaches has more focused on probabilistic contexts such as MDPs in the recent years. The PDDL language was extended to *probabilistic* PDDL (Younes *et al.*, 2005) to standardize syntax, and the *international planning competition* (IPC) was extended in 2004 with a probabilistic track to encourage tackling these domains, to facilitate cross-fertilization between approaches, and to provide a standard benchmark. The algorithms and representations in this book contribute to this field, and in fact, some of the methods we describe, have entered the IPC in recent years (e.g. Hölldobler *et al.*, 2006; Fern *et al.*, 2006). An additional historical connection exists with ML approaches for planning (Zimmerman and Kambhampati, 2003). As many algorithms for first-order MDPs use various kinds of ML algorithms, and learning is essentially focused on heuristics (i.e. value functions), much cross-fertilization is possible here.

Summary of Dimensions. As described, the material in this book extends three distinct fields in AI, see also Figure 1.7. First, it extends RL at a representational level by moving to first-order world representations. Second, it extends ML in two ways: one by adding a utility component to probabilistic, logical ML approaches, and another one by extending probabilistic logic learning to the RL learning paradigm. Third, it extends first order (probabilistic) planning approaches with a utility component, thereby extending propositional probabilistic planning approaches towards first-order knowledge representation.

Many questions¹⁴ naturally arise from the descriptions of the historical developments. For example, when is it possible to design algorithms for rich representations by reduction to traditional techniques? This is one of the leading questions when we go over from Chapters 2 and 3 to the rest of the chapters. Another question is about how RL can benefit from (or contribute to) existing models and techniques used for (decision-theoretic) planning and agents that already use richer representations, but lack learning? This is a question that will be approached in Chapter 4. Yet another one is about whether the interaction between rich representations and the (known and validated) framework of (PO)MDPs can be characterized in a theoretically rigorous way? This question is leading in many of

¹⁴Some of these questions played a central role in the workshop on 'rich representations for reinforcement learning' that we organized at the international conference on machine learning (ICML) in 2005.

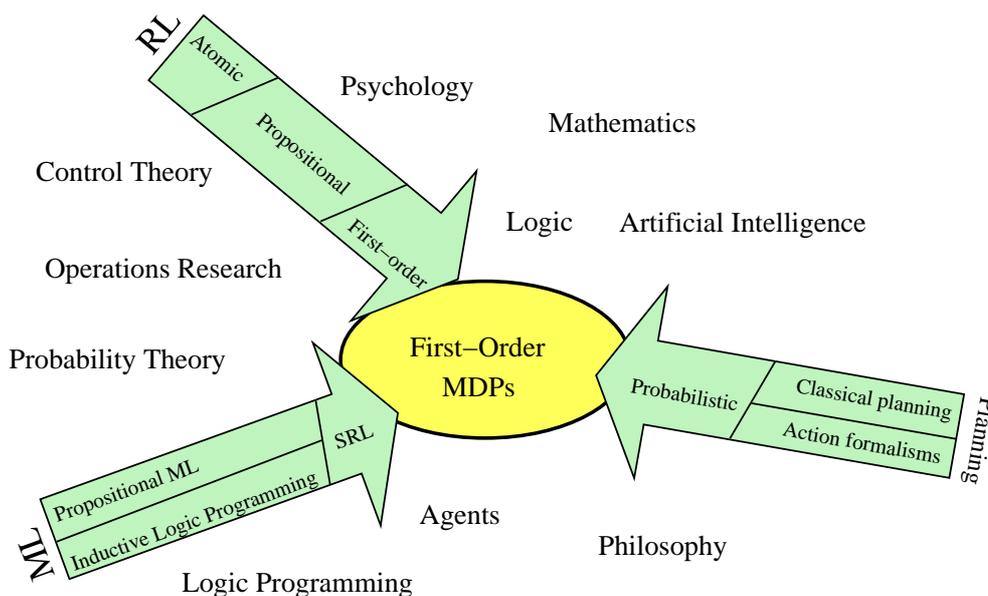


Figure 1.7: The material in this book embedded in subfields of AI.

the chapters, and in the final chapter we come back to this when identifying some of the remaining challenges.

1.3.2 A Road Map

The structure of this book closely follows the historical lines of the field of RL. In the first part we try to identify important concepts and algorithms that exist in the propositional setting. The second and third parts of the book are about the first-order setting. In the following we briefly describe each of the chapters.

1.3.2.1 PART I: LEARNING SEQUENTIAL DECISION MAKING UNDER UNCERTAINTY

The first part of the book deals with solving MDPs. This part will introduce the standard framework, highlight important methodological directions and review various methods. An important aspect of this part of the book is that it will describe various ways in which *abstraction* and *generalization* can be employed in the framework of MDPs and algorithms that compute solutions. Abstraction and generalization are studied in the propositional setting, and important concepts and techniques that are found will play an important role when we upgrade to the first-order knowledge representation setting in the second part of this book. This part can be seen as describing the *adaptive behavior* part of the title and providing answers to the first series of questions.

How can adaptive behavior be modeled and computed using the MDP framework, which types of efficient solution algorithms exist and how can propositional generalization and abstraction be employed in the framework?

This part consists of two chapters:

Chapter 2: Markov Decision Processes: Concepts and Algorithms

In this chapter, **the classical MDP framework** is described mathematically. This is the setting CANTOR operates in. States and actions have no structure and here we deal with the *algorithmic* part of learning sequential decision making. We distinguish two main types of algorithms for computing optimal policies. **Model-based** methods assume a known transition and reward model and employ DP techniques to obtain policies. **Model-free** methods interact with a domain simulator and use sampled traces to compute value functions and policies. For both types of algorithms we explain **various extensions** to make computations more efficient. This chapter lays down the mathematical foundation for much of the rest of the book, and some of the boundaries of the material are identified in the context of **non-Markovian** models and algorithms.

Chapter 3: Generalization and Abstraction in Markov Decision Processes

This chapter is about employing **abstraction and generalization** in **propositional** MDPs, which correspond to BOOLE's twenty-questions setting. Among the first things discussed is a description of what abstraction and generalization is, and why it is needed. Based on a distinction between fixed and adaptive representations, we introduce the novel **PIAGET principle**, as an extension of the general mechanism of *generalized policy iteration* (GPI) introduced by Sutton and Barto (1998). The principle provides a useful way to classify various algorithms and to see how behavior learning and representation formation interact. A large part of this chapter describes **five main types of abstraction**, that differ in the type of structures generalization is performed over. These are, in order, state spaces, transition and reward models (so-called factored representations), value functions, policies and hierarchical decompositions. For each of these five types, we survey important concepts, algorithms and methods, guided by the PIAGET principle. Described as a case study in value function approximation in RL we then outline a **novel RL application in fingerprint recognition** in which various types of concepts are combined. This chapter also includes a table which holds pointers to sections and chapters in this book where first-order versions of the algorithms that were discussed, can be found. The chapter ends with a discussion of the interplay between behavior and representation.

1.3.2.2 PART II: SEQUENTIAL DECISIONS IN THE FIRST-ORDER SETTING

The second part of the book deals with problems posed as a Markov decision process over a first-order domain, which is the environment we have described for the robot FREGE. It will describe the intuitions and arguments that together form the main motivation for moving to more powerful representation languages. The chapters in this part of the book form the main contributions to the field. We will distinguish between *model-free* and *model-based*, as was done in the previous two chapters. For both types of methods new algorithms will be described, and in addition, a full survey of existing methods is provided. This part can be seen as describing the *logic* element of the title of this book, and providing answers to the second series of questions.

What are first-order knowledge representation, abstraction, generalization and action formalisms, how can they be used for first-order versions of MDPs and how can propositional algorithms be upgraded to solve such MDPs?

This part of the book consists of three chapters.

Chapter 4: Reasoning, Learning and Acting in Worlds with Objects

This chapter defines the setting for our most advanced robot FREGE. We begin this chapter by defining why representing the world in terms of objects and relations is desired and necessary, why it is natural and what the consequences are. Based on our findings in the previous two chapters, we need three main things. First we describe **first-order knowledge representation and reasoning** and based on this we introduce the notion of a **relational Markov decision process (RMDP)**. A second aspect is **first-order abstraction and generalization** techniques. We describe inductive ML techniques for the first-order setting and survey important concepts and algorithms. A third main component is a definition of **first-order domain and action theories**. We describe a number of important issues in modeling dynamic domains, and some action logics that are helpful in later chapters. The second part of this chapter introduces a general **first-order represented MDP (FORM)** capturing the idea of specifying a relational MDP in any particular logical formalism. Additionally, we upgrade value functions, policies and the PIAGET principle to the first-order setting and discuss both the representational and algorithmic aspects in this new setting.

Placed in between Chapters 2 and 3 on the one hand, and Chapters 5 to 7 on the other, Chapter 4 functions as a *bridge* between propositional and first-order MDP approaches.

Chapter 5: Model-Free Algorithms for Relational MDPs

This chapter upgrades the **model-free** setting introduced in Chapters 2 and 3 to the first-order setting defined in Chapter 4, utilizing much of the first-order generalization techniques in that chapter. The first part of the chapter is about **value function approximation** methods. We introduce **CARCASS**, a **novel representational formalism for RMDP** and a model-free (Q -learning) algorithm for learning RMDP policies. Furthermore, we describe an indirect RL algorithm based on prioritized sweeping to compute value functions and policies more efficiently, by learning a transition model of the abstract RMDP. This part also contains an **extensive survey of all model-free, value function approximation methods for RMDPs**. The second part is about **policy search methods**. We introduce the first **evolutionary policy search for RMDPs**, named **GREY**. In addition, we provide an **extensive survey of all other policy search techniques for RMDPs**.

Chapter 6: Model-Based Algorithms for Relational MDPs

Complementing the previous chapter, this chapter is about the **model-based** setting in which the action formalisms defined in Chapter 4 play an important role. In this chapter we show how virtually all kinds of the model-based algorithms discussed in Chapters 2 and 3 can be unified in a **novel technique called intensional dynamic programming**. In five steps we move from classical value iteration to dynamic programming with general knowledge representation formalisms. As an intermediate step, we outline **set-based dynamic programming** that provides the semantics for intensional dynamic programming. In this way, an unified approach for many existing techniques for DP in the face of abstraction and generalization is provided. It is shown that it also applies to the first-order setting, and as

an example we introduce **REBEL**, the first implemented first-order dynamic programming approach. REBEL uses extensions of action formalisms and deductive techniques from Chapter 4 and can be applied even in problems with infinite domains. It is shown that DP in first-order domains introduces a number of new concepts and furthermore, REBEL clearly shows that in the relational setting, convergence of algorithms such as value iteration is not guaranteed anymore. We also introduce and analyze several **extensions** to REBEL, concerning logical and RL efficiency issues, and the use of background knowledge and bottom-up generalization for policy induction. At the end of the chapter we provide a **thorough survey of all existing model-based techniques for RMDPs**.

1.3.2.3 PART III: IMPLICATIONS, CHALLENGES AND CONCLUSIONS

The third part of this book goes beyond the approaches described in Chapters 5 and 6 and is concerned with models and behavioral decompositions as in hierarchical RL. Furthermore, this part outlines a number of distinct areas with open research questions, gaps in our current practical and theoretical understanding of applying decision-theoretic approaches in first-order domains. The main question in this part can be stated as

What are the implications of representations and algorithms for first-order MDPs for modular behaviors, models and knowledge transfer, and what are the main challenges in first-order MDPs?

The third part of the book deals with the *implications* of the new methods for RL in relational domains. There are prominent implications and possibilities for logical agents and logical agent programming languages. Furthermore, a number of new challenges arises. On the one hand, these challenges are about extending more of traditional RL methods to the relational domains. On the other hand, these are challenges because of the new possibilities due to the powerful combination of logic, probability, utility and ML. This part can be seen as providing answers to the third question, and it contains two chapters.

Chapter 7: Sapience, Models and Hierarchy

In Chapter 7 we describe how the ideas within relational RL might be taken as step further, by incorporating adaptivity in logical, cognitive agent architectures, so-called **sapient agents**. This involves **hierarchical decompositions** of behaviors and dealing with **transition models**. Furthermore, we discuss issues in **guidance** to help the learning agent, and **transfer** of learned knowledge to other, similar, problems. This chapter contains a **survey** of all such approaches that have been described in the literature.

Chapter 8: Conclusions and Future Directions

In this chapter we **reflect** on what has been accomplished in this book, and it contains the main **conclusions**. An important aspect of this chapter is that it outlines a number of research **challenges** that are interesting to pursue. These include technical advances such as lifting more propositional algorithms to the first-order case, new directions for first-order representations such as POMDPs, and conceptual challenges such as dealing rigorously with varying domain sizes and knowledge transfer.

1.3.3 Other Main Themes and Contributions

Two other main contributions this book makes, are:

A Reference Guide on Knowledge Representation in Reinforcement Learning

Representation is the central issue in this book. Chapter 3 contains an extensive exposition of abstraction and generalization dimensions in the context of MDPs and solution algorithms. As far as we know, this is the first time that all current major directions in RL have been brought together in one text. Throughout the book we highlight how abstraction and generalization techniques can be (and have been) upgraded to the relational representation case.

A Complete Survey of Relational Reinforcement Learning

One of the main contributions of this book is the **first complete survey of the field of relational RL**. Chapter 4 starts the exposition with an outline of relational representational formalisms and their use in modeling MDPs. The next three chapters (5–7) cover the complete spectrum of all methods that have been proposed in the literature. Chapter 8, finally, covers a number of challenges and future directions for the field.

The following sections together provide a complete survey of first-order approaches in RL.

Section 4.1.3.3 **From Propositional to Relational**. This section contains methods that cope with first-order domains by using propositional representations and algorithms. Among these are propositionalization approaches and deictic representations.

Section 5.3 **A Survey of Model-Free, Value-Based Approaches**. This part of the book describes methods for relational MDPs focusing on learning value functions. We distinguish between fixed and adaptive generalization.

Section 5.5 **A Survey of Policy-Based Model-Free Relational RL**. This part of the book describes methods for relational MDPs focusing on learning policies directly. We distinguish between policy search techniques based on evolutionary algorithms and techniques that use classification learning.

Section 6.5 **A Survey of Model-Based Approaches**. In this section we survey all methods that operate under the assumption that a full model of the RMDP is available. Among these are various exact algorithms, approximate versions of exact methods, and other approximate methods that, for example, upgrade solutions obtained in small, ground problem instances. This section also describes a number of approaches that deviate from the Markov assumption.

Section 7.3 **A Survey of Hierarchies, Models, Guidance and Transfer**. In this part of the book we survey a number of other approaches in first-order domains. These are hierarchical approaches, model-learning approaches and transfer techniques. Furthermore, some examples of decision-theoretic (agent) programming languages are discussed.

Section 8.2 **Future Challenges**. We highlight several directions in which the field of relational RL can be extended.

Part I

Learning Sequential Decision Making under Uncertainty

CHAPTER 2

Markov Decision Processes: Concepts and Algorithms

Situated in between supervised learning and unsupervised learning, the paradigm of reinforcement learning deals with learning in sequential decision making problems in which there is limited feedback. This chapter introduces the intuitions and concepts behind Markov decision processes and two classes of algorithms for computing optimal behaviors: reinforcement learning and dynamic programming. First the formal framework of Markov decision process is defined, accompanied by the definition of value functions and policies. The main part of this chapter deals with introducing foundational classes of algorithms for learning optimal behaviors, based on various definitions of optimality with respect to the goal of learning sequential decisions. Additionally, it surveys efficient extensions of the foundational algorithms, differing mainly in the way feedback given by the environment is used to speed up learning, and in the way they concentrate on relevant parts of the problem. For both model-based and model-free settings these efficient extensions have shown useful in scaling up to larger problems. The last part of the chapter briefly describes the more complex setting of partially observable Markov decision processes.

MARKOV DECISION PROCESSES (MDP) (Puterman, 1994) are an intuitive and fundamental formalism for *decision-theoretic planning* (DTP) (Boutilier *et al.*, 1999; Boutilier, 1999), reinforcement learning (RL) (Kokar and Reveliotis, 1993; Kaelbling *et al.*, 1996; Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998) and other learning problems in stochastic domains. In this model, an environment is modeled as a set of states and actions can be performed to control the system's state. The goal is to control the system in such a way that some performance criterium is maximized. Many problems such as (stochastic) planning problems, learning robot control and game playing problems have successfully been modeled in terms of an MDP. In fact, MDPs have become the *de facto* standard formalism for learning sequential decision making.

DTP (Boutilier *et al.*, 1999), e.g. planning using decision-theoretic notions to represent uncertainty and plan quality, is an important extension of the AI *planning paradigm*, adding the ability to deal with *uncertainty* in action effects and the ability to deal with less-defined *goals*. Furthermore it adds a significant dimension in that it considers situations in which factors such as resource consumption and uncertainty demand solutions of *varying quality*, for example in *real-time* decision situations. There are many connections between AI

planning, research done in the field of *operations research* (Winston, 1991) and *control theory* (Bertsekas, 1995), since most work in these fields are about *sequential decision making* and can be seen as operating over MDPs. The notion of a *plan* in AI planning, i.e. a series of actions from a start state to a goal state, is extended to the notion of a *policy*, which is mapping from *all* states to an (optimal) action, based on decision-theoretic measures of *optimality* with respect to some goal to be optimized.

As an example, consider a typical planning domain, involving boxes to be moved around and where the goal is to move some particular boxes to a designated area. This type of problems can be solved using AI planning techniques. Consider now a slightly more realistic extension in which some of the actions can fail, or have uncertain side-effects that can depend on factors beyond the operator's control, and where the goal is specified by giving credit for how many boxes are put on the right place. In this type of environment, the notion of a plan is less suitable, because a sequence of actions can have many different outcomes, depending on the effects of the operators used in the plan. Instead, the methods in this chapter are concerned about *policies* that map states onto actions in such a way that the *expected* outcome of the operators will have the intended effects. The expectation over actions is based on a decision-theoretic expectation with respect to their probabilistic outcomes and credits associated with the problem goals. The MDP framework allows for online solutions that *learn* optimal policies gradually through *simulated trials*, and additionally, it allows for *approximated* solutions with respect to resources such as computation time. Finally, the model allows for numeric, decision-theoretic measurement of the *quality* of policies and learning *performance*. For example, policies can be ordered by how much credit they receive, or by how much computation is needed for a particular performance.

This chapter will cover the broad spectrum of methods that have been developed in the literature to compute good or optimal policies for problems modeled as an MDP. The term RL is associated with the more difficult setting in which no (prior) knowledge about the MDP is presented. The task then of the algorithm is to *interact*, or *experiment* with the environment (i.e. the MDP), in order to gain knowledge about how to optimize its behavior, being guided by the evaluative feedback (rewards). The model-based setting, in which the full transition dynamics and reward distributions are known, is usually characterized by the use of *dynamic programming* (DP) techniques. However, we will see that the underlying basis is very similar, and that mixed forms occur.

The methods in this chapter cover the *discrete, finite* model case in which all states, actions etc. are stored as *discrete symbols*. The purpose of this exposition is to highlight important concepts, algorithms and directions in the field of learning sequential decision making. For most realistic problems, a shift towards *knowledge representation, abstraction* and *approximation* is needed, but we will defer discussion of these issues to Chapter 3. The current chapter will not contain a complete technical survey of all methods that have been proposed in the literature. Instead we focus on important directions and give a representative sample of the literature in each direction. The purpose of this chapter is threefold. First, it describes the learning setting used in this book. Second, it provides insights in algorithms, efficient versions of these algorithms and some motivations for using *knowledge representation* schemes in the MDP context, which we will describe in Chapter 3. Last, the *relational* and *first-order* methods described in the second half of this book will borrow heavily from the methods in this chapter.

environment	You are in state 65. You have 4 possible actions.
agent	I'll take action 2.
environment	You have received a reward of 7 units. You are now in state 15. You have 2 possible actions.
agent	I'll take action 1.
environment	You have received a reward of -4 units. You are now in state 65. You have 4 possible actions.
agent	I'll take action 2.
environment	You have received a reward of 5 units. You are now in state 44. You have 5 possible actions.
...	...

Figure 2.1: Example of an *agent-environment interaction*, from an RL perspective.

Outline. The next section will first introduce the main learning problem: *sequential decision making*. After that we formalize the problem using the framework of MDPs in Section 2.2. Most algorithms for MDPs make use of *value functions* for estimating policy quality and we will introduce them in Section 2.3. These value functions form the foundation for many model-based and model-free algorithms. We will describe model-based algorithms (DP) in Section 2.5 and model-free algorithms (RL) in Section 2.6. For both classes of algorithms several efficient improvements exist and we describe these in Section 2.5.2 and Section 2.6.3. Besides *value-based* methods, some *policy search* algorithms exist and we will describe these in Section 3.7. A more general model of sequential decision making is the *partially observable* MDP (POMDP). POMDPs extend the MDP by allowing that some of the environment state cannot be perceived, such that there is additional uncertainty. We describe some of the basics of POMDPs in Section 2.7. This chapter ends with a discussion in Section 2.8 which will review a number of important concepts encountered in this chapter.

2.1. Learning Sequential Decision Making

RL is a general class of algorithms in the field of machine learning that aims at allowing an *agent* to learn how to behave in an environment, where the only feedback consists of a *scalar* reward signal. RL should not be seen as characterized by a particular class of learning methods, but rather as a *learning problem* or a *paradigm*. The *goal* of the agent is to perform *actions* that maximize the reward signal in the long run.

The distinction between the *agent* and the *environment* might not always be the most intuitive one. We will draw a boundary based on *control* (see Sutton and Barto, 1998). Everything the agent cannot control, is considered part of the environment. For example, although the motors of a robot agent might be considered part of the agent, the exact functioning of them in the environment is beyond the agent's control. It can give commands to gear up or down, but their physical realization can be influenced by many things.

An example of interaction with the environment is given in Figure 2.1. It shows how the interaction between an *agent* and the *environment* can take place. The agent can choose an action in each state, and the *perceptions* the agent gets from the environment, are the environment's state after each action plus the scalar reward signal at each step. Here a

discrete model is used in which there are distinct numbers for each state and action. The way the interaction is depicted is highly general in the sense that one just talks about states and actions as discrete *symbols*. In the rest of this book we will be more concerned about interactions in which states and actions have more *structure*, such that a state can be something like *there are two blue boxes and one white one and you are standing next to a blue box*. However, this figure clearly shows the *mechanism* of sequential decision making.

There are several important aspects in learning sequential decision making which we will describe in this section, after which we will describe formalizations.

Approaching Sequential Decision Making. There are several classes of algorithms that deal with the problem of sequential decision making. In this book we deal specifically with the topic of *learning*, but some other options exist.

The first solution is the *programming* solution. An intelligent system for sequential decision making can – in principle – be *programmed* to handle all situations. For each possible state an appropriate or optimal action can be specified *a priori*. However, this puts a heavy burden on the designer or programmer of the system. All situations should be foreseen in the design phase and programmed into the agent. This is a tedious and almost impossible task for most interesting problems, and it only works for problems which can be modeled completely. In most realistic problems this is not possible due to the sheer *size* of the problem, or the intrinsic *uncertainty* in the system. A simple example is *robot control* in which factors such as lighting or temperature can have a large, and unforeseen, influence on the behavior of camera and motor systems. Furthermore, in situations where the problem changes, for example due to new elements in the description of the problem or changing dynamic of the system, a programmed solution will no longer work. Programmed solutions are *brittle* in that they will only work for completely known, static problems with fixed probability distributions. A second solution uses *search* and *planning* for sequential decision making. The successful CHESS program *Deep Blue* (Schaeffer and Plaat, 1997) was able to defeat the human world champion Gary Kasparov by smart, brute force search algorithms that used a model of the dynamics of CHESS, tuned to Kasparov's style of playing. When the dynamics of the system is known, one can *search* or *plan* from the current state to a desirable goal state. However, when there is uncertainty about the action outcomes standard search and planning algorithms do not apply. *Admissible heuristics* can solve some problems concerning the reward-based nature of sequential decision making, but the probabilistic effects of actions pose a difficult problem. Probabilistic planning algorithms exist (e.g. Kushmerick *et al.*, 1995), but their performance is not as good as their deterministic counterparts. An additional problem is that planning and search focus on specific start and goal states. In contrast, we are looking for *policies* which are defined for all states, and are defined with respect to *rewards*.

The third solution is *learning*, and this will be the main topic of this book. Learning has several advantages in sequential decision making. First, it relieves the designer of the system from the difficult task of deciding upon everything in the design phase. Second, it can cope with uncertainty, goals specified in terms of reward measures, and with changing situations. Third, it is aimed at solving the problem for every state, as opposed to a mere plan from one state to another. Additionally, although a model of the environment can be used or learned, it is not necessary in order to compute optimal policies, such as is exemplified by RL methods. Everything can be learned from interaction with the environment.

Online versus Off-line Learning. One important aspect in the learning task we consider in this book, is the distinction between *online* and *off-line* learning. The difference between these two types is influenced by factors such as whether one wants to control a *real-world* entity – such as a robot player robot soccer or a machine in a factory – or whether all necessary information is available. Online learning performs learning directly on the problem instance. Off-line learning uses a *simulator* of the environment as a cheap way to get many training examples for *safe* and *fast* learning.

Learning the controller directly on the real task is often not possible. For example, the learning algorithms in this chapter sometimes need millions of training instances which can be too time-consuming to collect. Instead, a simulator is much faster, and in addition it can be used to provide arbitrary training situations, including situations that rarely happen in the real system. Furthermore, it provides a “safe” training situation in which the agent can explore and make mistakes. Obtaining negative feedback in the real task in order to learn to avoid these situations, might entail destroying the machine that is controlled, which is unacceptable. Often one uses a simulation to obtain a reasonable policy for a given problem, after which some parts of the behavior are *fine-tuned* on the real task. For example, a simulation might provide the means for learning a reasonable robot controller, but some *physical* factors concerning variance in motor and perception systems of the robot might make additional fine-tuning necessary. A simulation is just a *model* of the real problem, such that small differences between the two are natural, and learning might make up for that difference. Many problems in the literature however, *are* simulations of games and optimization problems, such that the distinction disappears.

Credit Assignment. An important aspect of sequential decision making is the fact that deciding whether an action is “good” or “bad” cannot be decided upon right away. The appropriateness of actions is completely determined by the *goal* the agent is trying to pursue. The real problem is that the effect of actions with respect to the goal can be much *delayed*. For example, the opening moves in CHESS have a large influence on winning the game. However, between the first opening moves and receiving a reward for winning the game, a couple of tens of moves might have been played. Deciding how to give *credit* to the first moves – which did not get the immediate reward for winning – is a difficult problem called the *temporal credit assignment* problem. Each move in a winning CHESS game contributes more or less to the success of the last move, although some moves along this path can be less optimal or even bad. A related problem is the *structural credit assignment* problem, in which the problem is to distribute feedback over the *structure* representing the agent’s policy. For example, the policy can be represented by a structure containing parameters (e.g. a neural network). Deciding which parameters have to be updated forms the structural credit assignment problem. In this chapter we will assume that all representations are explicit tables such that this problem is of less importance, but in the next chapter we will encounter various *structured* representations in which we will have to deal with this problem as well.

The Exploration-Exploitation Trade-off. If we know a complete model of dynamics of the problem, there exist methods (e.g. DP) that can compute optimal policies from this model. However, in the more general case where we do not have access to this knowledge (e.g. RL), it becomes necessary to *interact* with the environment to learn by *trial-and-error* a correct policy. The agent has to *explore* the environment by performing actions

and perceiving their consequences (i.e. the effects on the environments and the obtained rewards). The only feedback the agent gets are rewards, but it does not get information about what is the right action. At some point in time, it will have a policy with a particular performance. In order to see whether there are possible improvements to this policy, it sometimes has to *try out* various actions to see their results. This might result in worse performance because the actions might also be less good than the current policy. However, without trying them, it might never find possible improvements. In addition, if the world is not stationary, the agent has to explore to keep its policy up-to-date. So, in order to *learn* it has to *explore*, but in order to *perform well* it should *exploit* what it already knows. Balancing these two things is called the *exploration-exploitation problem*.

Feedback, Goals and Performance. Compared to supervised learning, the amount of feedback the learning system gets in RL, is much less. In supervised learning, for every learning sample the correct output is given in a training set. The performance of the learning system can be measured relative to the number of correct answers, resulting in a *predictive accuracy*. The difficulty lies in learning this mapping, and whether this mapping *generalizes* to new, unclassified, examples. In unsupervised learning, the difficulty lies in constructing a useful partitioning of the data such that classes naturally arise. In RL there is only some information available about performance, in the form of one *scalar* signal. This feedback system is *evaluative* rather than being *instructive*. Using this limited signal for feedback renders a need to put more effort in using it to evaluate and improve behavior during learning.

A second aspect about feedback and performance is related to the stochastic nature of the problem formulation. In supervised and unsupervised learning, the data is usually considered *static*, i.e. a data set is given and performance can be measured with respect to this data. The learning samples for the learner originate from a fixed distribution, i.e. the data set. From an RL perspective, the data can be seen as a *moving target*. The learning process is driven by the current policy, but this policy will change over time. That means that the *distribution* over states and rewards will change because of this. In machine learning the problem of a changing distribution of learning samples is termed *concept drift* (Maloof, 2003) and it demands special features to deal with it. In RL this problem is dealt with by exploration, a constant interaction between evaluation and improvement of policies and additionally the use of *learning rate adaption schemes*.

A third aspect of feedback is the question "*where do the numbers come from?*". In many sequential decision tasks, suitable reward functions present themselves quite naturally. For games in which there are winning, losing and draw situations, the reward function is easy to specify. In some situations special care has to be taken in giving rewards for states or actions, and also their *relative size* is important. When the agent will encounter a large negative reward before it finally gets a small positive reward, this positive reward might get *overshadowed*. All problems posed will have *some* optimal policy, but it depends on whether the reward function is in accordance with the right goals, whether the policy will tackle the *right* problem. In some problems it can be useful to provide the agent with rewards for reaching intermediate *subgoals*. This can be helpful in problems which require very long action sequences.

Representations. One of the most important aspects in learning sequential decision making is *representation*. Two central issues are *what* should be represented, and *how*

things should be represented. The first issue is dealt with in this chapter. Key components that can or should be represented, are models of the dynamics of the environment, reward distributions, value functions and policies. For some algorithms all components are explicitly stored in tables, for example in classical DP algorithms. Actor-critic methods keep separate, explicit representations of both value functions and policies. However, in most RL algorithms just a value function is represented whereas policy decisions are derived from this value function online. Methods that search in policy space (see Section 3.7) do not represent value functions explicitly, but instead an explicitly represented policy is used to compute values when necessary. Overall, the choice for *not* representing certain elements can influence the choice for a type of algorithm, and its efficiency.

The question of *how* various structures can be represented is dealt with extensively in this book, starting from the next chapter. Structures such as policies, transition functions and value functions can be represented in more compact form by using various *structured* knowledge representation formalisms and this enables much more efficient solution mechanisms and scaling up to larger domains.

Psychological Background and History. Many of the conceptual ideas in technical, computational work in RL come from psychology, and more specifically *behaviorism*. Famous researchers such as Skinner¹ (1904–1990, operant conditioning), Pavlov (1849–1936, the classical conditioning paradigm) and Thorndike (1874–1949, reinforcement theories) have studied adaptive processes in animal and human learning and many of their results have – more or less – been implemented in computational systems in the recent decades. The core mechanism of RL can be stated as *the Law of Effect*:

”Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond.”

Thorndike (1911, p.244).

Although most work in current RL has focused on formal models such as Markov decision processes and various extensions thereof, some work has considered more brain-like models that incorporate RL (e.g. see Rivest *et al.*, 2004).

This book is about the technical aspects of RL, and more specifically about representation. We cannot do justice to the large amount of literature that is available on the history of RL and related fields, also because other texts can do that in much greater detail. Much can still be learned from the psychological side to create new models, concepts and algorithms. On the other hand, psychological and brain theories might benefit from computational approaches to verify and test different types of learning in restricted settings. We refer to (Dayan and Abbott, 2001; Dayan, 2000; Alonso and Mondragón, 2006; Jozefowicz, 2002; Witkowski, 2007) and (Bakker, 2004, par.3.2.2) and furthermore to (Witkowski, 1997, chap.2) and (Drescher, 1991) for interesting discussions of general

¹Interestingly, Skinner also wrote a novel *Walden II* about a society based on his research findings.

(psychological) matters in RL and related fields, and (Sutton and Barto, 1998) for historical pointers, for example the early work done by Samuel (1959) on CHECKERS, Minsky, Michie and Holland.

2.2. A Formal Framework

The elements of the RL problem as described in the introduction to this chapter can be formalized using the *Markov decision process* (MDP) framework. In this section we will formally describe components such as *states* and *actions* and *policies*, as well as the *goals* of learning using different kinds of *optimality criteria*. MDPs are extensively described in (Puterman, 1994) and (Boutilier *et al.*, 1999). They can be seen as stochastic extensions of finite automata and also as *Markov processes* augmented with actions.

Although general MDPs may have infinite (even uncountable) state and action spaces, we limit the discussion to finite-state and finite-action problems. In the next chapter we will encounter continuous spaces and in later chapters we will encounter situations arising in the first-order logic setting in which infinite spaces can quite naturally occur.

2.2.1 Markov Decision Processes.

MDPs consist of states, actions, transitions between states and a reward function definition. We consider each of them in turn.

States. The set of environmental states S is defined as the finite set $\{s^1, \dots, s^N\}$ where the *size* of the state space is N , i.e. $|S| = N$. A *state* is a unique characterization of all that is important in a state of the problem that is modeled. For example, in CHESS a complete configuration of board pieces of both black and white, is a state. In the next chapter we will encounter the use of *features* that *describe* the state. In those contexts, it becomes necessary to distinguish between *legal* and *illegal* states, for some combinations of features might not result in an actually existing state in the problem. In this chapter, we will confine ourselves to the discrete state set S in which each state is represented by a *distinct symbol*, and all states $s \in S$ are legal.

Actions. The set of actions A is defined as the finite set $\{a^1, \dots, a^K\}$ where the *size* of the action space is K , i.e. $|A| = K$. Actions can be used to *control* the system state. The set of actions that can be applied in some particular state $s \in S$, is denoted $A(s)$, where $A(s) \subseteq A$. In some systems, not all actions can be applied in every state, but in general we will assume that $A(s) = A$ for all $s \in S$. In more structured representations (e.g. by means of *features*), the fact that some actions are not applicable in some states, is modeled by a *precondition function* $\text{pre} : S \times A \rightarrow \{\text{true}, \text{false}\}$, stating whether action $a \in A$ is applicable in state $s \in S$.

The Transition Function. By applying action $a \in A$ in a state $s \in S$, the system makes a *transition* from s to a new state $s' \in S$, based on a probability distribution over the set of possible transitions. The transition function T is defined as $T : S \times A \times S \rightarrow [0, 1]$, i.e. the probability of ending up in state s' after doing action a in state s is denoted $T(s, a, s')$. It is required that for all actions a , and all states s and s' , $T(s, a, s') \geq 0$ and $T(s, a, s') \leq 1$. Furthermore, for all states s and actions a , $\sum_{s' \in S} T(s, a, s') = 1$, i.e. T defines a *proper probability distribution over possible next states*. Instead of a precondition function, it is

also possible to set² $T(s, a, s') = 0$ for all states $s' \in S$ if a is not applicable in s . For talking about the *order* in which actions occur, we will define a discrete *global clock*, $t = 1, 2, \dots$. Using this, the notation s_t denotes the state at time t and s_{t+1} denotes the state at time $t + 1$. This enables to compare different states (and actions) occurring ordered in time during interaction. The system being controlled is *Markovian* if the result of an action does not depend on the previous actions and visited states (history), but only depends on the current state, i.e.

$$P(s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, \dots) = P(s_{t+1} \mid s_t, a_t) = T(s_t, a_t, s_{t+1})$$

The idea of Markovian dynamics is that the current state s gives enough information to make an optimal decision; it is not important which states and actions preceded s . Another way of saying this, is that if you select an action a , the probability distribution over next states is the same as the last time you tried this action in the same state. More general models can be characterized by being *k-Markov*, i.e. the last k are states sufficient, such that *Markov* is actually *1-Markov*. Though, each *k-Markov* problem can be transformed into an equivalent *Markov* problem. The *Markov property* forms a boundary between the MDP and more general models such as POMDPs (see Section 2.7 for a brief description).

The Reward Function. The *reward function*³ specifies rewards for being in a state, or doing some action in a state. The *state reward* function is defined as $R : S \rightarrow \mathbb{R}$, and it specifies the reward obtained in states. However, two other definitions exist. One can define either $R : S \times A \rightarrow \mathbb{R}$ or $R : S \times A \times S \rightarrow \mathbb{R}$. The first one gives rewards for performing an action in a state, and the second gives rewards for particular transitions between states. All definitions are interchangeable though the last one is convenient in *model-free* algorithms (see Section 2.6), because there we usually need both the starting state and the resulting state in backing up values. Throughout this book we will mainly use $R(s, a, s')$, but deviate from this when more convenient.

The reward function is an important part of the MDP that specifies implicitly the *goal* of learning. For example, in episodic tasks such as in the games TIC-TAC-TOE and CHESS, one can assign all states in which the agent has won a positive reward value, all states in which the agent loses a negative reward value and a zero reward value in all states where the final outcome of the game is a draw. The goal of the agent is to reach positive valued states, which means winning the game. Thus, the reward function is used to give *direction* in which way the system, i.e. the MDP, should be controlled. Often, the reward function assigns non-zero reward to non-goal states as well, which can be interpreted as defining *sub-goals* for learning.

The Markov Decision Process. Putting all elements together results in the definition of a *Markov decision process*, which will be the base model for the large majority of methods described in this book.

²Although this is the same, the explicit distinction between an action not being applicable in a state and a zero probability for transitions with that action, is lost in this way.

³Although we talk about *rewards* here, with the usual connotation of something positive, the reward function merely gives a *scalar* feedback signal. This can be interpreted as negative (*punishment*) or positive (*reward*). The various origins of work in MDPs in the literature creates an additional confusion with the reward function. In the *operations research* literature, one usually speaks of a *cost function* instead and the goal of learning and optimization is to *minimize* this function.

DEFINITION 2.2.1 ▶ A **Markov decision process** is a tuple $\langle S, A, T, R \rangle$ in which S is a finite set of states, A a finite set of actions, T a transition function defined as $T : S \times A \times S \rightarrow [0, 1]$ and R a reward function defined as $R : S \times A \times S \rightarrow \mathbb{R}$.

The transition function T and the reward function R together define the *model* of the MDP. Often MDPs are depicted as a state transition graph (see Figure 3.4 for an example) where the nodes correspond to states and (directed) edges denote transitions. A typical domain that is frequently used in the MDP literature is the *maze* (Matthews, 1922), in which the reward function assigns a positive reward for reaching the exit state.

There are several distinct types of systems that can be modeled by this definition of an MDP. In *episodic tasks*, there is the notion of *episodes* of some length, where the goal is to take the agent from a starting state to a *goal state*. An *initial state distribution* $I : S \rightarrow [0, 1]$ gives for each state the probability of the system being started in that state. Starting from a state s the system progresses through a sequence of states, based on the actions performed. In episodic tasks, there is a specific subset $G \subseteq S$, denoted *goal state area* containing states (usually with some distinct reward) where the process *ends*. We can furthermore distinguish between *finite, fixed horizon* tasks in which each episode consists of a fixed number of steps, *indefinite horizon* tasks in which each episode can end but episodes can have arbitrary length, and *infinite horizon* tasks where the system does not end at all. The last type of model is usually called a *continuing task*.

Episodic tasks, i.e. in which there so-called *goal states*, can be modeled using the same model defined in Definition 2.2.1. This is usually modeled by means of *absorbing states* or *terminal states*, e.g. states from which every action results in a transition to that same state with probability 1 and reward 0. Formally, for an absorbing state s , it holds that $T(s, a, s) = 1$ and $R(s, a, s') = 0$ for all states $s' \in S$ and actions $a \in A$. When entering an absorbing state, the process is reset and restarts in a new starting state. Episodic tasks and absorbing states can in this way be modeled in the same framework as continuing tasks.

2.2.2 Policies

Given an MDP $\langle S, A, T, R \rangle$, a policy is a computable function that outputs for each state $s \in S$ an action $a \in A$ (or $a \in A(s)$). Formally, a *deterministic policy* π is a function defined as $\pi : S \rightarrow A$. It is also possible to define a *stochastic policy* as $\pi : S \times A \rightarrow [0, 1]$ such that for each state $s \in S$, it holds that $\pi(s, a) \geq 0$ and $\sum_{a \in A} \pi(s, a) = 1$. We will assume deterministic policies in this book unless stated otherwise.

Application of a policy to an MDP is done in the following way. First, a start state s_0 from the initial state distribution I is generated. Then, the policy π suggest the action $a_0 = \pi(s_0)$ and this action is performed. Based on the transition function T and reward function R , a transition is made to state s_1 , with probability $T(s_0, a, s_1)$ and a reward $r_0 = R(s_0, a_0, s_1)$ is received. This process continues, producing $s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, \dots$. If the task is episodic, the process ends in state s_{goal} and is restarted in a new state drawn from I . If the task is continuing, the sequence of states can be extended indefinitely.

The policy is part of the agent and its aim is to *control* the environment modeled as an MDP. A fixed policy induces a stationary transition distribution over the MDP which can be transformed into a *Markov system*⁴ $\langle S', T' \rangle$ where $S' = S$ and $T'(s, s') = T(s, a, s')$

⁴In other words, if π is fixed, the system behaves as a stochastic transition system with a stationary distribution over states.

whenever $\pi(s) = a$.

2.2.3 Optimality Criteria and Discounting

In the previous sections, we have defined the environment (the MDP) and the agent (i.e. the controlling element, or policy). Before we can talk about algorithms for computing *optimal* policies, we have to define what that means. That is, we have to define what the *model of optimality* is. There are two ways of looking at optimality. First, there is the aspect of *what* is actually being optimized, i.e. what is the *goal* of the agent? Second, there is the aspect of *how* optimal the way in which the goal is being optimized, is. The first aspect is related to *gathering reward* and is treated in this section. The second aspect is related to the efficiency and optimality of algorithms, and this is briefly touched upon and dealt with more extensively in Section 2.4 and further.

The goal of learning in an MDP is to gather rewards. If the agent was only concerned about the immediate reward, a simple optimality criterion would be to optimize $E[r_t]$. However, there are several ways of taking into account the future in how to behave now. There are basically three models of optimality in the MDP, which are sufficient to cover most of the approaches in the literature. They are strongly related to the types of tasks that were defined in Section 2.2.1.

$$E \left[\sum_{t=0}^h r_t \right] \qquad E \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \qquad \lim_{h \rightarrow \infty} E \left[\frac{1}{h} \sum_{t=0}^h r_t \right]$$

Figure 2.2: Optimality: a) finite horizon, b) discounted, infinite horizon, c) average reward.

The *finite horizon* model simply takes a finite horizon of length h and states that the agent should optimize its expected reward over this horizon, i.e. the next h steps (see Figure 2.2a). One can think of this in two ways. The agent could in the first step take the *h -step optimal action*, after this the *$(h - 1)$ -step optimal action*, and so on. Another way is that the agent will always take the *h -step optimal action*, which is called *receding-horizon control*. The problem, however, with this model, is that the (optimal) choice for the horizon length h is not always known.

In the *infinite-horizon model*, the long-run reward is taken into account, but the rewards that are received in the future are discounted according to how far away in time they will be received. A *discount factor* γ , with $0 \leq \gamma < 1$ is used for this (see Figure 2.2b). Note that in this *discounted* case, rewards obtained later are discounted more than rewards obtained earlier. Additionally, the discount factor ensures that – even with infinite horizon – the sum of the rewards obtained is finite. In episodic tasks, i.e. in tasks where the horizon is finite, the discount factor is not needed or can equivalently be set to 1. If $\gamma = 0$ the agent is said to be *myopic*, which means that it is only concerned about immediate rewards. The discount factor can be interpreted in several ways; as an interest rate, probability of living another step, or the mathematical trick for bounding the infinite sum. The discounted, infinite-horizon model is mathematically more convenient, but conceptually similar to the finite horizon model. Most algorithms in this book use this model of optimality.

A third optimality model is the *average-reward* model, maximizing the long-run *average reward* (see Figure 2.2c). Sometimes this is called the *gain optimal* policy and, as the discount factor approaches 1, it can be seen as the limiting case of the infinite-horizon dis-

counted model. A difficult problem with this criterion is that we cannot distinguish between two policies in which one receives a lot of reward in the initial phases and another one which does not. This initial difference in reward is hidden in the long-run average. This problem can be solved in using a *bias optimal* model in which the long-run average is still being optimized, but policies are preferred if they additionally get initially extra reward. See (Mahadevan, 1996) for a survey on average reward RL.

Choosing between these optimality criteria can be related to the learning problem. If the length of the episode is known, the finite-horizon model is best. However, often this is not known, or the task is continuing, the infinite-horizon model is more suitable. Koenig and Liu (2002) give an extensive overview of different modelings of MDPs and their relationship with optimality.

The second kind of optimality in this section is related to the more general aspect of the optimality of the learning process itself. We will encounter various concepts in the remainder of this book. We will briefly summarize three important notions here.

Learning optimality can be explained in terms of *what* the end result of learning might be. A first concern is whether the agent is able to obtain *optimal performance* in principle. For some algorithms there are proofs stating this, but for some not. In other words, is there a way to ensure that the learning process will reach a global optimum, or merely a local optimum, or even an oscillation between performances? A second kind of optimality is related to the *speed* of converging to a solution. We can distinguish between two learning methods by looking at how many interactions are needed, or how much computation is needed per interaction. And related to that, what will the performance be after a certain period of time? In supervised learning the optimality criterion is often defined in terms of *predictive accuracy* which is different from optimality in the MDP setting. Also, it is important to look at how much *experimentation* is necessary, or even allowed, for reaching optimal behavior. For example, a learning robot or helicopter might not be allowed to make many mistakes during learning. A last kind of optimality is related to how much reward is *not* obtained by the learned policy, as compared to an optimal one. This is usually called the *regret* of a policy.

2.3. Value Functions and Bellman Equations

In the preceding sections we have defined MDPs and optimality criteria that can be useful for learning optimal policies. In this section we define *value functions* (see Bartlett, 2003), which are a way to link the optimality criteria to policies. Most learning algorithms for MDPs compute optimal policies by learning value functions. A value function represents an estimate of *how good* it is for the agent to be in a certain state (or how good it is to perform a certain action in that state). The notion of *how good* is expressed in terms of an optimality criterion, i.e. in terms of the expected return. Value functions are defined for particular policies.

The *value of a state s under policy π* , denoted $V^\pi(s)$ is the expected return when starting in s and following π thereafter. We will use the infinite-horizon, discounted model in this

section, such that this can be expressed⁵ as:

$$V^\pi(s) = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s \right\} \quad (2.1)$$

A similar *state-action value function* $Q : S \times A \rightarrow \mathbb{R}$ can be defined as the expected return starting from state s , taking action a and thereafter following policy π :

$$Q^\pi(s, a) = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, a_t = a \right\}$$

One fundamental property of value functions is that they satisfy certain recursive properties. For any policy π and any state s the expression in Equation 2.1 can recursively be defined in terms of a so-called *Bellman Equation* (Bellman, 1957):

$$\begin{aligned} V^\pi(s) &= E_\pi \left\{ r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \mid s_t = s \right\} \\ &= E_\pi \left\{ r_t + \gamma V^\pi(s_{t+1}) \mid s_t = s \right\} \\ &= \sum_{s'} T(s, \pi(s), s') \left(R(s, \pi(s), s') + \gamma V^\pi(s') \right) \end{aligned} \quad (2.2)$$

It denotes that the expected value of state is defined in terms of the immediate reward and values of possible next states weighted by their transition probabilities, and additionally a discount factor. V^π is the unique solution for this set of equations. Note that multiple policies can have the same value function, but for a given policy π , V^π is unique.

The goal for any given MDP is to find a *best* policy, i.e. the policy that receives the most reward. This means maximizing the value function of Equation 2.1 for all states $s \in S$. An *optimal policy*, denoted π^* , is such that $V^{\pi^*}(s) \geq V^\pi(s)$ for all $s \in S$ and all policies π . It can be proved that the optimal solution $V^* = V^{\pi^*}$ satisfies the following Equation:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s, a, s') \left(R(s, a, s') + \gamma V^*(s') \right) \quad (2.3)$$

This expression is called the *Bellman optimality equation*. It states that the value of a state under an optimal policy must be equal to the expected return for the best action in that state. To select an optimal action given the optimal state value function V^* the following rule can be applied:

$$\pi^*(s) = \arg \max_a \sum_{s' \in S} T(s, a, s') \left(R(s, a, s') + \gamma V^*(s') \right) \quad (2.4)$$

We call this policy the *greedy policy*, denoted $\pi_{greedy}(V)$ because it greedily selects the best action using the value function V . An analogous optimal state-action value is:

$$Q^*(s, a) = \sum_{s'} T(s, a, s') \left(R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right)$$

⁵Note that we use E_π for the *expected value under policy* π .

Q -functions are useful because they make the weighted summation over different alternatives (such as in Equation 2.4) using the transition function unnecessary. No forward-reasoning step is needed to compute an optimal action in a state. This is the reason that in model-free approaches, i.e. in case T and R are unknown, Q -functions are learned instead of V -functions. The relation between Q^* and V^* is given by

$$Q^*(s, a) = \sum_{s' \in S} T(s, a, s') \left(R(s, a, s') + \gamma V^*(s') \right) \quad (2.5)$$

$$V^*(s) = \max_a Q^*(s, a) \quad (2.6)$$

Now, analogously to Equation 2.4, optimal action selection can be simply put as:

$$\pi^*(s) = \arg \max_a Q^*(s, a) \quad (2.7)$$

That is, the best action is the action that has the highest expected utility based on possible next states resulting from taking that action. One can, analogously to the expression in Equation 2.4, define a greedy policy $\pi_{greedy}(Q)$ based on Q . In contrast to $\pi_{greedy}(V)$ there is no need to consult the model of the MDP; the Q -function suffices.

2.4. Solving Markov decision processes

Now that we have defined MDPs, policies, optimality criteria and value functions, it is time to consider the question of *how* to compute optimal policies. *Solving* a given MDP means computing an optimal policy π^* . Several dimensions exist along which algorithms have been developed for this purpose. The most important distinction is that between *model-based* and *model-free* algorithms.

Model-based algorithms exist under the general name of DP. The basic assumption in these algorithms is that a *model* of the MDP is known beforehand, and can be used to compute value functions and policies using the Bellman equation (see Equation 2.3). Most methods are aimed at computing state value functions which can, in the presence of the model, be used for optimal action selection. In this chapter we will focus on *iterative* procedures for computing value functions and policies.

Model-free algorithms, under the general name of RL, do not rely on the availability of a perfect model. Instead, they rely on *interaction* with the environment, i.e. a *simulation* of the policy thereby generating *samples* of state transitions and rewards. These samples are then used to estimate state-action value functions. Because a model of the MDP is not known, the agent has to *explore* the MDP to obtain information. This naturally induces a *exploration-exploitation* trade-off which has to be balanced to obtain an optimal policy.

A very important underlying mechanism, the so-called *generalized policy iteration* (GPI) principle, present in all methods is depicted in Figure 2.3. This principle consists of two interacting processes. The *policy evaluation* step estimates the utility of the current policy π , that is, it computes V^π . There are several ways for computing this. In model-based algorithms, one can use the model to compute it directly or iteratively approximate it. In model-free algorithms, one can *simulate* the policy and estimate its utility from the sampled execution traces. The main purpose of this step is to gather information about the policy for computing the second step, the *policy improvement* step. In this step, the values of the actions are evaluated for every state, in order to find possible improvements, i.e.

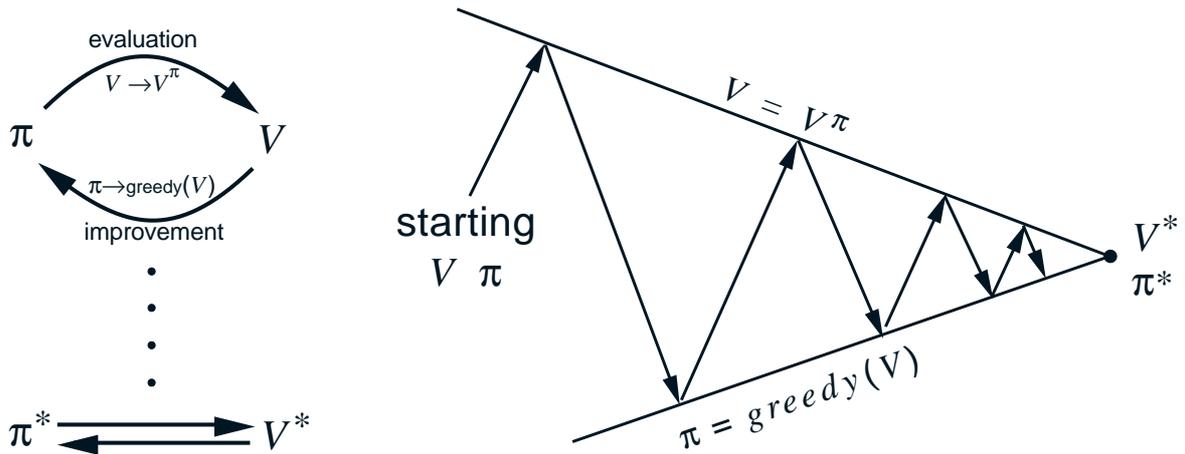


Figure 2.3: a) The algorithms in Section 2.4 can be seen as instantiations of **Generalized Policy Iteration (GPI)** (picture adapted from Sutton and Barto, 1998). The policy evaluation step estimates V^π , the policy’s performance. The policy improvement step improves the policy π based on the estimates in V^π . b) The gradual convergence of both the value function and the policy to optimal versions.

possible other actions in particular states that are better than the action the current policy proposes. This step computes an improved policy π' from the current policy π using the information in V^π . Both the evaluation and the improvement steps can be implemented in various ways, and interleaved in several distinct ways. The bottom line is that there is a policy that drives value learning, i.e. it determines the value function, but in turn there is a value function that can be used by the policy to select good actions. Note that it is also possible to have an *implicit* representation of the policy, which means that only the value function is stored, and a policy is computed on-the-fly for each state based on the value function when needed. This is common practice in model-free algorithms (see Section 2.6). And vice versa it is also possible to have implicit representations of value functions in the context of an explicit policy representation. Another interesting aspect is that in general a value function does not have to be perfectly accurate. In many cases it suffices that sufficient distinction is present between suboptimal and optimal actions, such that small errors in values do not have to influence policy optimality. This is also important in *approximation* and *abstraction* methods discussed in the next chapter.

Planning as an RL Problem. The MDP formalism is a general formalism for *decision-theoretic planning*, which entails that standard (deterministic) *planning* problems can be formalized as such too. All the algorithms in this chapter can – in principle – be used for these planning problems too. In order to solve planning problems in the MDP framework we have to specify *goals* and *rewards*. We can assume that the transition function T is given, accompanied by a *precondition function*. In planning we are given a *goal function* $G : S \rightarrow \{\text{true}, \text{false}\}$ that defines which states are goal states. The planning task is compute a sequence of actions $a_t, a_{t+1}, \dots, a_{t+n}$ such that applying this sequence from a *start* state will lead to a state $s \in G$. All transitions are assumed to be deterministic, i.e. for all states $s \in S$ and actions $a \in A$ there exists only one state $s' \in S$ such that $T(s, a, s') = 1$. All states in G are assumed to be absorbing. The only thing left is to specify the reward function. We can specify this in such a way that a positive reinforcement is received once

a goal state is reached, and zero otherwise:

$$R(s_t, a_t, s_{t+1}) = \begin{cases} 1, & \text{if } s_t \notin G \text{ and } s_{t+1} \in G \\ 0, & \text{otherwise} \end{cases}$$

Now, depending on whether the transition function and reward function are known to the agent, one can solve this planning task with either model-based or model-free learning. The difference with classical planning is that the learned policy will apply to all states.

2.5. Dynamic Programming: Model-Based Solution Techniques

The term DP refers to a class of algorithms that is able to compute optimal policies in the presence of a perfect model of the environment. The assumption that a model is available will be hard to ensure for many applications. However, we will see that from a theoretical viewpoint, as well as from an algorithmic viewpoint, DP algorithms are very relevant because they define fundamental computational mechanisms which are also used when no model is available. The methods in this section all assume a standard MDP $\langle S, A, T, R \rangle$, where the state and action sets are finite and discrete such that they can be stored in tables. Furthermore, transition, reward and value functions are assumed to store values for all states and actions separately.

2.5.1 Fundamental DP Algorithms

Two core DP methods are *policy iteration* (Howard, 1960) and *value iteration* (Bellman, 1957). In the first, the GPI mechanism is clearly separated into two steps, whereas the second represents a tight integration of policy evaluation and improvement. We will consider both these algorithms in turn.

2.5.1.1 POLICY ITERATION

Policy iteration (PI) (Howard, 1960) iterates between the two phases of GPI. The *policy evaluation* phase computes the value function of the current policy and the *policy improvement* phase computes an improved policy by a maximization over the value function. This is repeated until converging to an optimal policy.

Policy Evaluation: The Prediction Problem. A first step is to find the value function V^π of a fixed policy π . This is called the *prediction problem*. It is a part of the complete problem, that of computing an *optimal* policy. Remember from the previous sections that for all $s \in S$,

$$V^\pi(s) = \sum_{s' \in S} T(s, \pi(s), s') \left(R(s, \pi(s), s') + \gamma V^\pi(s') \right) \quad (2.8)$$

If the dynamics of the system is known, i.e. a model of the MDP is given, then these equations form a system of $|S|$ equations in $|S|$ unknowns (the values of V^π for each $s \in S$). This can be solved by linear programming (LP). However, an *iterative* procedure is possible, and in fact common in DP and RL. The Bellman equation is transformed into an *update rule* which updates the current value function V_k^π into V_{k+1}^π by 'looking one step

further in the future', thereby extending the planning horizon with one step:

$$\begin{aligned} V_{k+1}^\pi(s) &= E_\pi \left\{ r_t + \gamma V_k^\pi(s_{t+1}) \mid s_t = s \right\} \\ &= \sum_{s'} T(s, \pi(s), s') \left(R(s, \pi(s), s') + \gamma V_k^\pi(s') \right) \end{aligned} \quad (2.9)$$

The sequence of approximations of V_k^π as k goes to infinity can be shown to converge. In order to converge, the update rule is applied to each state $s \in S$ in each iteration. It replaces the old value for that state by a new one that is based on the expected value of possible successor states, intermediate rewards and weighted by the transition probabilities. This operation is called a *full backup* because it is based on all possible transitions from that state.

A more general formulation can be given by defining a *backup operator* B^π over arbitrary real-valued functions φ over the state space (e.g. a value function):

$$(B^\pi \varphi)(s) = \sum_{s' \in S} T(s, \pi(s), s') \left(R(s, \pi(s), s') + \gamma \varphi(s') \right) \quad (2.10)$$

The value function V^π of a fixed policy π satisfies the *fixed point* of this backup operator as $V^\pi = B^\pi V^\pi$. A useful special case of this backup operator is defined with respect to a fixed action a :

$$(B^a \varphi)(s) = R(s) + \gamma \sum_{s' \in S} T(s, a, s') \varphi(s')$$

Now LP for solving the *prediction problem* can be stated as follows. Computing V^π can be accomplished by solving the Bellman equations (see Equation 2.3) for all states. The *optimal* value function V^* can be found by using a LP problem solver that computes $V^* = \arg \max_V \sum_s V(s)$ subject to $V(s) \leq (B^a V)(s)$ for all a and s .

Policy Improvement. Now that we know the value function V^π of a policy π as the outcome of the policy evaluation step, we can try to improve the policy. First we identify the value of all actions by using:

$$Q^\pi(s, a) = E_\pi \left\{ r_t + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a \right\} \quad (2.11)$$

$$= \sum_{s'} T(s, a, s') \left(R(s, a, s') + \gamma V^\pi(s') \right) \quad (2.12)$$

If now $Q^\pi(s, a)$ is larger than $V^\pi(s)$ for some $a \in A$ then we could do better by choosing action a instead of the current $\pi(s)$. In other words, we can *improve* the current policy by selecting a different, better, action in a particular state. In fact, we can evaluate all actions in all states and choose the best action in all states. That is, we can compute the *greedy* policy π' by selecting the best action in each state, based on the current value function V^π :

$$\begin{aligned} \pi'(s) &= \arg \max_a Q^\pi(s, a) \\ &= \arg \max_a E \left\{ r_t + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a \right\} \\ &= \arg \max_a \sum_{s'} T(s, a, s') \left(R(s, a, s') + \gamma V^\pi(s') \right) \end{aligned} \quad (2.13)$$

Algorithm 1 Policy Iteration (Howard, 1960).

Require: $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$

```

1: % POLICY EVALUATION
2: repeat
3:    $\Delta := 0$ 
4:   for each  $s \in S$  do
5:      $v := V^\pi(s)$ 
6:      $V(s) := \sum_{s'} T(s, \pi(s), s') \left( R(s, \pi(s), s') + \gamma V(s') \right)$ 
7:      $\Delta := \max(\Delta, |v - V(s)|)$ 
8:   until  $\Delta < \sigma$ 
9: % POLICY IMPROVEMENT
10: policy-stable := true
11: for each  $s \in S$  do
12:    $b := \pi(s)$ 
13:    $\pi(s) := \arg \max_a \sum_{s'} T(s, a, s') \left( R(s, a, s') + \gamma V(s') \right)$ 
14:   if  $b \neq \pi(s)$  then policy-stable := false
15: if policy-stable then stop; else go to POLICY EVALUATION
    
```

Computing an improved policy by greedily selecting the best action with respect to the value function of the original policy is called *policy improvement*. If the policy cannot be improved in this way, it means that the policy is already optimal and its value function satisfies the Bellman equation for the optimal value function. In a similar way one can also perform these steps for stochastic policies by blending the action probabilities into the expectation operator.

Summarizing, *policy iteration* (Howard, 1960) starts with an arbitrary initialized policy π_0 . Then a sequence of iterations follows in which the current policy is evaluated after which it is improved. The first step, the *policy evaluation* step computes V^{π_k} , making use of Equation 2.9 in an iterative way. The second step, the *policy improvement* step, computes π_{k+1} from π_k using V^{π_k} . For each state, using equation 2.4, the optimal action is determined. If for all states s , $\pi_{k+1}(s) = \pi_k(s)$, the policy is *stable* and the policy iteration algorithm can stop. Policy iteration generates a sequence of alternating policies and value functions

$$\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \rightarrow \pi_2 \rightarrow V^{\pi_2} \rightarrow \pi_3 \rightarrow V^{\pi_3} \rightarrow \dots \rightarrow \pi^*$$

The complete algorithm can be found in Algorithm 1.

For finite MDPs, i.e. state and action spaces are finite, policy iteration converges after a finite number of iterations. Each policy π_{k+1} is a strictly better policy than π_k unless in case $\pi_k = \pi^*$, in which case the algorithm stops. And because for a finite MDP, the number of different policies is finite, policy iteration converges in finite time. In practice, it usually converges after a small number of iterations. Although policy iteration computes the optimal policy for a given MDP in finite time, it is relatively inefficient. In particular the first step, the policy evaluation step, is computationally expensive. Value functions for all intermediate policies $\pi_0, \dots, \pi_k, \dots, \pi^*$ are computed, which involves multiple sweeps through the complete state space per iteration. A bound on the number of iterations is difficult to characterize (see Littman *et al.*, 1995; Mansour and Singh, 1999) and depends

Algorithm 2 Value Iteration (Bellman, 1957).

Require: initialize V arbitrarily (e.g. $V(s) := 0, \forall s \in S$)

```

1: repeat
2:    $\Delta := 0$ 
3:   for each  $s \in S$  do
4:      $v := V(s)$ 
5:     for each  $a \in A(s)$  do
6:        $Q(s, a) := \sum_{s'} T(s, a, s') \left( R(s, a, s') + \gamma V(s') \right)$ 
7:      $V(s) := \max_a Q(s, a)$ 
8:      $\Delta := \max(\Delta, |v - V(s)|)$ 
9: until  $\Delta < \sigma$ 
    
```

on the MDP transition structure, but it often converges after few iterations in practice.

2.5.1.2 VALUE ITERATION

The policy iteration algorithm completely separates the evaluation and improvement phases. In the evaluation step, the value function must be computed in the limit. However, it is not necessary to wait for full convergence, but it is possible to stop evaluating earlier and improve the policy based on the evaluation so far. The extreme point of truncating the evaluation step is the *value iteration* (Bellman, 1957) algorithm. It breaks off evaluation after just one iteration. In fact, it immediately blends the policy improvement step into its iterations, thereby purely focusing on estimating directly the value function. Necessary updates are computed on-the-fly. In essence, it combines a truncated version of the policy evaluation step with the policy improvement step, which is essentially Equation 2.3 turned into one update rule:

$$V_{t+1}(s) = \max_a \sum_{s'} T(s, a, s') \left(R(s, a, s') + \gamma V_t(s') \right) \quad (2.14)$$

$$= \max_a Q_{t+1}(s, a). \quad (2.15)$$

Using Equations (2.14) and (2.15), the value iteration algorithm (see Algorithm 2) can be stated as follows: starting with a value function V_0 over all states, one iteratively updates the value of each state according to (2.14) to get the next value functions V_t ($t = 1, 2, 3, \dots$). It produces the following sequence of value functions:

$$V_0 \rightarrow V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_5 \rightarrow V_6 \rightarrow V_7 \rightarrow \dots V^*$$

Actually, in the way it is computed it also produces the intermediate Q -value functions such that the sequence is

$$V_0 \rightarrow Q_1 \rightarrow V_1 \rightarrow Q_2 \rightarrow V_2 \rightarrow Q_3 \rightarrow V_3 \rightarrow Q_4 \rightarrow V_4 \rightarrow \dots V^*$$

Value iteration is guaranteed to converge in the limit towards V^* , i.e. the Bellman optimality Equation (2.3) holds for each state. A deterministic policy π for all states $s \in S$ can be computed using Equation 2.4. If we use the same general *backup operator* mechanism

used in the previous section, we can define value iteration in the following way.

$$(B^*\varphi)(s) = \max_a \sum_{s' \in S} T(s, a, s') \left\{ R(s, a, s') + \gamma\varphi(s) \right\} \quad (2.16)$$

The backup operator B^* functions as a *contraction mapping* on the value function. If we let π^* denote the *optimal* policy and V^* its value function, we have the relationship (fixed point) $V^* = B^*V^*$ where $(B^*V)(s) = \max_a (B^aV)(s)$. If we define $Q^*(s, a) = B^aV^*$ then we have $\pi^*(s) = \pi_{greedy}(V^*)(s) = \arg \max_a Q^*(s, a)$. That is, the algorithm starts with an arbitrary value function V_0 after which it iterates $V_{t+1} = B^*V_t$ until $\|V_{t+1} - V_t\|_S < \epsilon$, i.e. until the distance between subsequent value function approximations is small enough. We shall return to this topic in Chapter 6.

2.5.2 Efficient DP Algorithms

The policy iteration and value iteration algorithms can be seen as spanning a *spectrum* of DP approaches. This spectrum ranges from complete *separation* of evaluation and improvement steps to a complete *integration* of these steps. Clearly, in between the extreme points is much room for variations on algorithms, and in addition also those that *parallelize* computation (e.g. see Wingate and Seppi, 2005). Let us first consider the computational complexity of the extreme points.

Complexity. Value iteration works by producing successive approximations of the optimal value function. Each iteration can be performed in $O(|A||S|^2)$ steps, or faster if T is sparse. However, the *number* of iterations can grow exponentially in the discount factor (see Bertsekas and Tsitsiklis, 1996). This follows from the fact that a larger γ implies that a longer sequence of future rewards has to be taken into account, hence a larger number of value iteration steps because each step only extends the horizon taking into account in V by one step. The complexity of value iteration is linear in number of actions, and quadratic in the number of states. But, usually the transition matrix is sparse. In practice policy iteration converges much faster, but each evaluation step is expensive. Each iteration has a complexity of $O(|A||S|^2 + |S|^3)$, which can grow large quickly. A rough worst case bound on the number of iterations can be given (e.g. see Littman *et al.*, 1995; Mansour and Singh, 1999). Linear programming is a common tool that can be used for the evaluation too. In general, the number of iterations and value backups can quickly grow extremely large when the problem size grows. The state spaces of games such as backgammon and CHESS consist of too many states to perform just one full sweep. In this section we will describe some efficient variations on DP approaches. Detailed coverage of complexity results for the solution of MDPs can be found in (Littman *et al.*, 1995; Bertsekas and Tsitsiklis, 1996; Boutilier *et al.*, 1999).

The efficiency of DP can be roughly improved along two lines. The first is a *tighter integration* of the evaluation and improvement steps of the GPI process. We will discuss this issue briefly in the next section. The second is that of using (*heuristic*) search algorithms in combination with DP algorithms. For example, using search as an exploration mechanism can highlight important parts of the state space such that value backups can be concentrated on these parts. This is the underlying mechanism used in the methods discussed briefly in Section 2.5.2.2

2.5.2.1 STYLES OF UPDATING

The full backup updates in DP algorithms can be done in several ways. We have assumed in the description of the algorithms that in each step an old and a new value function are kept in memory. Each update puts a new value in the new table, based on the information of the old. This is called *synchronous*, or *Jacobi-style* updating (Sutton and Barto, 1998). This is useful for explanation of algorithms and theoretical proofs of convergence. However, there are two more common ways for updates. One can keep a single table and do the updating directly in there. This is called *in-place* updating (Sutton and Barto, 1998) or *Gauss-Seidel* (Bertsekas and Tsitsiklis, 1996) and usually speeds up convergence, because during one sweep of updates, some updates use already newly updated values of other states. Another type of updating is called *asynchronous updating* which is an extension of the *in-place* updates, but here updates can be performed in any order. An advantage is that the updates may be distributed unevenly throughout the state(-action) space, with more updates being given to more important parts of this space. For all these methods convergence can be proved under the general condition that values are updated infinitely often but with a finite frequency.

Modified Policy Iteration. *Modified policy iteration* (MPI) (Puterman and Shin, 1978) strikes a middle ground between value and policy iteration. MPI maintains the two separate steps of GPI, but both steps are not necessarily computed in the limit. The key insight here is that for policy improvement, one does not need an *exactly* evaluated policy in order to improve it. For example, the policy estimation step can be approximative after which a policy improvement step can follow. In general, both steps can be performed quite independently *by different means*. For example, instead of iteratively applying the Bellman update rule from Equation 2.15, one can perform the policy estimation step by using a sampling procedure such as *Monte Carlo* estimation (Sutton and Barto, 1998). These general forms in which mixed forms of estimation and improvements is captured by the generalized policy iteration mechanism depicted in Figure 2.3. Policy iteration and value iteration are both the extreme cases of modified policy iteration, whereas MPI is a general method for asynchronous updating.

2.5.2.2 HEURISTICS AND SEARCH

In many realistic problems, only a fraction of the state space is relevant to the problem of reaching the goal state from some state s . This has inspired a number of algorithms that focus computation on states that seem most relevant for finding an optimal policy from a start state s . These algorithms usually display good *anytime behavior*, i.e. they produce good or reasonable policies fast, after which they are gradually improved. In addition, they can be seen as implementing various ways of *asynchronous* DP.

Envelopes and Fringe States. One form of *asynchronous* methods is the PLEXUS system (Dean *et al.*, 1995). It was designed for goal-based reward functions, i.e. episodic tasks in which only goal states get positive reward. It starts with an approximated version of the MDP in which not the full state space is contained. This smaller version of the MDP is called an *envelope* and it includes the agent's current state and the goal state. A special OUT state represents all the states outside the envelope. The initial envelope is constructed by a forward search until a goal state is found. The envelope can be extended

by considering states outside the envelope that can be reached with high probability. The intuitive idea is to include in the envelope all the states that are likely to be reached on the way to the goal. Once the envelope has been constructed, a policy is computed through policy iteration. If at any point the agent leaves the envelope, it has to replan by extending the envelope. This combination of learning and planning still uses policy iteration, but on a much smaller (and presumably more relevant with respect to the goal) state space.

A related method proposed by Tash and Russell (1994) considers goal-based tasks too. However, instead of the single OUT state, they keep a *fringe* of states on the edge of the envelope and use a heuristic to estimate values of the other states. When computing a policy for the envelope, all fringe states become absorbing states with the heuristic set as their value. Over time the heuristic values of the fringe states converge to the optimal values of those states.

Similar to the previous methods the LAO* algorithm (Hansen and Zilberstein, 2001) also alternates between an expansion phase and a policy generation phase. It too keeps a fringe of states outside the envelope such that expansions can be larger than the envelope method by Dean *et al.* (1995). The motivation behind LAO* was to extend the classical search algorithm AO* (see Russell and Norvig, 2003) to *cyclic* domains such as MDPs.

Search and Planning in DP. *Real-time* DP (RTDP) by Barto *et al.* (1995) combines forward search with DP too. It is used as an alternative for value iteration in which only a subset of values in the state space are backed up in each iteration. RTDP performs *trials* from a randomly selected state to a goal state, by simulating the greedy policy using an *admissible heuristic* function as the initial value function. It then backups values *fully* only along these trials, such that backups are concentrated on the *relevant* parts of the state. The approach was later extended into *labeled* RTDP by Bonet and Geffner (2003b) where some states are *labeled* as *solved* which means that their value has already converged. Furthermore, it was recently extended to *bounded* RTDP by McMahan *et al.* (2005) which keeps lower and upper *bounds* on the optimal value function. Other recent methods along these lines are *focussed* DP (Ferguson and Stentz, 2004) and *heuristic search-DP* (Bonet and Geffner, 2003a)

2.6. Reinforcement Learning: Model-Free Solution Techniques

The previous section has reviewed several methods for computing an optimal policy for an MDP assuming that a (perfect) model is available. RL is primarily concerned with how to obtain an optimal policy when such a model is not available. RL adds to MDPs a focus on approximation and incomplete information, and the need for sampling and exploration. In contrast with the algorithms discussed in the previous section, *model-free* methods do not rely on the availability of priori known transition and reward models, i.e. a *model of the* MDP. The lack of a model generates a need to *sample* the MDP to gather statistical knowledge about this unknown model. Many model-free RL techniques exist that probe the environment by doing actions, thereby estimating the same kind of state value and state-action value functions as model-based techniques. This section will review model-free methods along with several efficient extensions.

In model-free contexts one has still a choice between two options. The first one is first to *learn* the transition and reward model from interaction with the environment. After that, when the model is (approximately or sufficiently) correct, all the DP methods from the

Algorithm 3 A general algorithm for online RL.

```

1: for each episode do
2:    $s \in S$  is initialized as the starting state
3:   repeat
4:     choose an action  $a \in A(s)$  using the current policy  $\pi$ 
5:     perform action  $a$ 
6:     observe the new state  $s'$  and received reward  $r$ 
7:     update  $\tilde{T}$ ,  $\tilde{R}$ ,  $\pi$ ,  $\tilde{Q}$  and/or  $\tilde{V}$ 
8:     using the experience  $\langle s, a, r, s' \rangle$ 
9:      $s := s'$ 
10:  until  $s'$  is a goal state OR maximum number of episodes reached

```

previous section apply. This type of learning is called *indirect* RL. The second option, called *direct* RL, is to step right into estimating values for actions, without even estimating the model of the MDP. Additionally, mixed forms between these two exists too. For example, one can still do model-free estimation of action values, but use an approximated model to speed up value learning by using this model to perform more, and in addition, full backups of values (see Section 2.6.3). Most model-free methods however, focus on direct estimation of (action) values.

A second choice one has to make is what to do with the *temporal credit assignment*. It is difficult to assess the utility of some action, if the real effects of this particular action can only be perceived much later. One possibility is to wait until the "end" (e.g. of an episode) and punish or reward specific actions along the path taken. However, this will take a lot of memory and often, with ongoing tasks, it is not known beforehand whether, or when, there will be an "end". Instead, one can use similar mechanisms as in *value iteration* to adjust the estimated value of a state based on the immediate reward and the estimated (discounted) value of the next state. This is generally called *temporal difference learning* which is a general mechanism underlying the model-free methods in this section. The main difference with the update rules for DP approaches (such as Equation 2.14) is that the transition function T and reward function R cannot appear in the update rules now. The general class of algorithms that interact with the environment and update their estimates after each experience is called *online*.

A general template for *online* RL is depicted in Algorithm 3. It shows an interaction loop in which the agent selects an action (by whatever means) based on its current state, gets feedback in the form of the resulting state and an associated reward, after which it updates its estimated values stored in \tilde{V} and \tilde{Q} and possibly statistics concerning \tilde{T} and \tilde{R} (in case of some form of indirect learning). The selection of the action is based on the current state s and the value function (either \tilde{Q} or \tilde{V}). To solve the exploration-exploitation problem, usually a separate *exploration* mechanism ensures that sometimes the best action (according to current estimates of action values) is taken (exploitation) but sometimes a different action is chosen (exploration). Various choices for exploration, ranging from random to sophisticated, exist and we will see some examples in Section 2.6.3.

Exploration. One important aspect of model-free algorithms is that there is a need for *exploration*. Because the model is unknown, the learner has to try out different actions to see their results. A learning algorithm has to strike a balance between *exploration* and

exploitation, i.e. in order to gain a lot of reward the learner has to exploit its current knowledge about good actions, although it sometimes must try out different actions to explore the environment for possible better actions. The most basic exploration strategy is the ϵ -greedy policy, i.e. the learner takes its current best action with probability $(1 - \epsilon)$ and a (randomly selected) other action with probability ϵ . There are many more ways of doing exploration (see Wiering, 1999; Reynolds, 2002; Ratitch, 2005, for overviews) and in Section 2.6.3 we will see some examples. One additional method that is often used in combination with the algorithms in this section is the *Boltzmann* (or: *softmax*) exploration strategy. It is only slightly more complicated than the ϵ -greedy strategy. The action selection strategy is still random, but selection probabilities are weighted by their relative Q -values. This makes it more likely for the agent to choose very good actions, whereas two actions that have similar Q -values will have almost the same probability to get selected. Its general form is

$$P(a_n) = \frac{e^{\frac{Q(s,a_n)}{T}}}{\sum_i e^{\frac{Q(s,a_i)}{T}}} \quad (2.17)$$

in which $P(a_n)$ is the probability of selecting action a_n and T is the *temperature* parameter. Higher values of T will move the selection strategy more towards a purely random strategy and lower values will move to a fully greedy strategy. A combination of both ϵ -greedy and Boltzmann exploration can be taken by taking the best action with probability $(1 - \epsilon)$ and otherwise an action computed according to Equation 2.17 (Wiering, 1999).

Another simple method to stimulate exploration is *optimistic Q -values initialization*; one can initialize all Q -values to high values – e.g. an a priori defined upper bound – at the start of learning. Because in this case Q -values will decrease during learning, actions that have not been tried a number of times will have a large enough value to get selected when using Boltzmann exploration for example. Another solution with a similar effect is to keep counters on the number of times a particular state-action pair has been selected.

2.6.1 Temporal Difference Learning

Temporal difference learning algorithms learn estimates of values based on other estimates. Each step in the world generates a *learning example* which can be used to bring some value in accordance to the immediate reward and the estimated value of the next state or state-action pair. An intuitive example, along the lines of (Sutton and Barto, 1998, Chapter 6), is the following.

Imagine you have to predict at what time your guests can arrive for a small diner in your house. Before cooking, you have to go to the supermarket, the butcher and the wine seller, in that order. You have estimates of driving times between all locations, and you predict that you can manage to visit the two last stores both in 10 minutes, but given the busy time on the day, your estimate about the supermarket is a half hour. Based on this prediction, you have notified your guests that they can arrive no earlier than 18.00h. Once you have found out while in the supermarket that it will take you only 10 minutes to get all the things you need, you can adjust your estimate on arriving back home with 20 minutes less. However, once on your way from the butcher to the wine seller, you see that there is quite some traffic along the way and it takes you 30 minutes longer to get there. Finally you arrive 10 minutes later than you predicted in the first place. The bottom line of this

example is that you can adjust your estimate about what time you will be back home every time you have obtained new information about in-between steps. Each time you can adjust your estimate on how long it will still take based on actually experienced times of parts of your path. This is the main principle of TD learning: you do not have to wait until the end of a trial to make updates along your path.

TD methods learn their value estimates based on estimates of other values, which is called *bootstrapping*. They have an advantage over DP in that they do not require a model of the MDP. Another advantage is that they are naturally implemented in an online, incremental fashion such that they can be easily used in various circumstances. No full sweeps through the full state space are needed; only along experienced paths values get updated, and updates are effected after each step.

TD(0). TD(0) is a member of the family of TD learning algorithms (Sutton, 1988). It solves the prediction problem, i.e. it estimates V^π for some policy π , in an online, incremental fashion. $TD(0)$ can be used to evaluate a policy and works through the use of the following update rule⁶:

$$V_{k+1}(s) = V_k(s) + \alpha \left(r + \gamma V_k(s') - V_k(s) \right)$$

where $\alpha \in [0, 1]$ is the *learning rate*, that determines by how much values get updated⁷. This backup is performed after experiencing the transition from state s to s' based on the action a , while receiving reward r . The difference with DP backups such as used in Equation 2.14 is that the update is still done by using bootstrapping, but it is based on an *observed* transition, i.e. it uses a *sample backup* instead of a full backup. Only the value of one successor state is used, instead of a weighted average of all possible successor states. When using the value function V^π for action selection, a model is needed to compute an expected value over all action outcomes (e.g. see Equation 2.4).

The learning rate α has to be decreased appropriately for learning to converge. Sometimes the learning rate can be defined for states separately as in $\alpha(s)$, in which case it can be dependent on how often the state is visited. The next two algorithms learn Q -functions directly from samples, removing the need for a transition model for action selection.

Q -learning. One of the most basic and popular methods to estimate Q -value functions in a model-free fashion, is the Q -learning algorithm by Watkins (1989); Watkins and Dayan (1992), see Algorithm 4.

The basic idea in Q -learning is to incrementally estimate Q -values for actions, based on feedback (i.e. rewards) and the agent's Q -value function. The update rule is a variation on the theme of TD learning, using Q -values and a built-in max-operator over the Q -values of the next state in order to update Q_t into Q_{t+1} :

$$Q_{k+1}(s, a) = Q_k(s, a) + \alpha \left(r + \gamma \max_{a'} Q_k(s', a') - Q_k(s, a) \right) \quad (2.18)$$

⁶The learning parameter α must comply with some criteria on its value, and the way it is changed. Most often a small, *fixed* learning parameter is chosen, or it is decreased every iteration.

⁷In some way, the same distinction can be found in probability theory. Full Bellman backups are like Bayesian conditioning, where the evidence is taken as the truth. Sample-based updating, i.e. Bellman backups along traces, are like Jeffrey conditioning, where the evidence is taken only up to a degree of belief (e.g. also see Gärdenfors, 1988).

Algorithm 4 Q-Learning (Watkins and Dayan, 1992).

Require: discount factor γ , learning parameter α

- 1: initialize Q arbitrarily (e.g. $Q(s, a) = 0, \forall s \in S, \forall a \in A$)
- 2: **for each** episode **do**
- 3: s is initialized as the *starting* state
- 4: **repeat**
- 5: choose an action $a \in A(s)$ based on an exploration strategy
- 6: perform action a
- 7: observe the new state s' and received reward r
- 8: $Q(s, a) := Q(s, a) + \alpha \left(r + \gamma \max_{a' \in A(s')} Q(s', a') - Q(s, a) \right)$
- 9: $s := s'$
- 10: **until** s' is a goal state

The agent makes a step in the environment from state s to s' using action a while receiving reward r . The update takes place on the Q -value of action a in the state s from which this action was executed.

Q-learning is exploration-insensitive. It means that it will converge to the optimal policy regardless of the exploration policy being followed, under the assumption that each state-action pair is visited an infinite number of times, and the learning parameter α is decreased appropriately (Watkins and Dayan, 1992; Bertsekas and Tsitsiklis, 1996).

SARSA. Q-learning is an *off-policy* learning algorithm, which means that while following some exploration policy π , it aims at estimating the optimal policy π^* . A related *on-policy* algorithm that learns the Q -value function for the policy the agent is actually executing is the SARSA (Rummery and Niranjan, 1994; Rummery, 1995; Sutton, 1996) algorithm, which stands for **State–Action–Reward–State–Action**. It uses the following update rule:

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha \left(r_t + \gamma Q_t(s', \pi(s')) - Q_t(s, a) \right) \quad (2.19)$$

where the action $\pi(s')$ is the one that is executed by the current policy for state s' . Note that the \max -operator in Q-learning is replaced by the estimate of the value of the next action according to the policy. This learning algorithm will still converge in the limit to the optimal value function (and policy) under the condition that all states and actions are tried infinitely often and the policy converges in the limit to the greedy policy, i.e. such that exploration does not occur anymore. SARSA is especially useful in non-stationary environments. In these situations one will never reach an optimal policy. It is also useful if *function approximation* is used, because off-policy methods can diverge when this is used. However, off-policy methods are needed in many situations such as in learning using hierarchically structured policies (see the next Chapter).

Actor-Critic Learning. Another class of algorithms that precede Q-learning and SARSA are *actor-critic* methods (Witten, 1977; Barto *et al.*, 1983; Konda and Tsitsiklis, 2003), which learn on-policy. This branch of TD methods keeps a *separate* policy independent of the value function. The policy is called the *actor* and the value function the *critic*. The critic – typically a state-value function – evaluates, or: criticizes, the actions executed by

the actor. After action selection, the critic evaluates the action using the TD-error:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

The purpose of this error is to strengthen or weaken the selection of this action in this state. A *preference* for an action a in some state s can be represented as $p(s, a)$ such that this preference can be modified using:

$$p(s_t, a_t) = p(s_t, a_t) + \beta \delta_t$$

where a parameter β determines the size of the update. There are other versions of actor-critic methods, differing mainly in how preferences are changed, or experience is used (for example using *eligibility traces*, see next section). An advantage of having separate policy representation is that if there are many actions, or when the action space is continuous, there is no need to consider all actions' Q -values in order to select one of them. A second advantage is that they can learn *stochastic* policies naturally. Furthermore, a priori knowledge about policy constraints can be used (e.g. see Främling, 2005).

Average Reward Temporal Difference Learning. We have explained Q -learning and related algorithms in the context of discounted, infinite-horizon MDPs. Q -learning can also be adapted to the average-reward framework, for example in the R -learning algorithm by Schwartz (1993). Other extensions of algorithms to the average reward framework exist (see Mahadevan, 1996, for an overview).

2.6.2 Monte Carlo Methods

Other algorithms that use more *unbiased* estimates are *Monte Carlo* (MC) techniques. They keep frequency counts of transitions and rewards and base their values on these estimates. MC methods only require samples to estimate average sample returns. For example, in MC policy evaluation, for each state $s \in S$ all returns obtained from s are kept and the value of a state $s \in S$ is just their average. In other words, MC algorithms treat the long-term reward as a random variable and take as its estimate the sampled mean. In contrast with one-step TD methods, MC estimates values based on *averaging sample returns* observed during interaction. Especially for episodic tasks this can be very useful, because samples from complete returns can be obtained. One way of using MC is by using it for the evaluation step in policy iteration. However, because the sampling is dependent on the current policy π , only returns for actions suggested by π are evaluated. Thus, exploration is of key importance here, just as in other model-free methods.

A distinction can be made between *every-visit* MC, which averages over all *visits* of a state $s \in S$ in all episodes, and *first-visit* MC, which averages over just the returns obtained from the first visit to a state $s \in S$ for all episodes. Both variants will converge to V^π for the current policy π over time. MC methods can also be applied to the problem of estimating action values. One way of ensuring enough exploration is to use *exploring starts*, i.e. each state-action pair has a non-zero probability of being selected as the initial pair. MC methods can be used for both on-policy and off-policy control, and the general pattern complies with the generalized policy iteration procedure. The fact that MC methods do not *bootstrap* makes them less dependent on the *Markov assumption*. TD methods too focus on *sampled* experience, although they do use bootstrapping.

Learning a Model. We have described MC methods in the context of learning value functions. Methods similar to MC can also be used to estimate a *model* of the MDP. An average over sample transition probabilities experienced during interaction can be used to gradually estimate transition probabilities. The same can be done for *immediate* rewards. *Indirect* RL algorithms make use of this to strike a balance between model-based and model-free learning. They are essentially model-free, but learn a transition and reward model in parallel with model-free RL, and use this model to do more efficient value function learning (see also the next section). An example of this is the DYNA model by Sutton (1991a). Another method that often employs model learning is *prioritized sweeping* (Moore and Atkeson, 1993). Learning a model can also be very useful to learn in continuous spaces where the transition model is defined over a discretized version of the underlying (infinite) state space (Großmann, 2000).

Relations with Dynamic Programming. The methods in this section solve essentially similar problems as DP techniques. RL approaches can be seen as *asynchronous* DP. There are some important differences in both approaches though.

RL approaches avoid the exhaustive sweeps of DP by restricting computation on, or in the neighborhood of, sampled trajectories, either real or simulated. This can exploit situations in which many states have low probabilities of occurring in actual trajectories. The backups used in DP are simplified by using sampling. Instead of generating and evaluating all of a state's possible immediate successors, the estimate of a backup's effect is done by sampling from the appropriate distribution. MC methods use this to base their estimates completely on the sample returns, without bootstrapping using values of other, sampled, states. Furthermore, the focus on learning (action) value functions in RL is easily amenable to *function approximation* approaches. Representing value functions and or policies can be done more compactly than lookup-table representations by using *numeric regression algorithms* without breaking the standard RL interaction process; one can just feed the update values into a regression engine. This topic will be covered more extensively in the next chapter.

An interesting point here is the similarity between Q -learning and *value iteration* on the one hand and SARSA and *policy iteration* on the other hand. In the first two methods, the updates immediately combine policy evaluation and improvement into one step by using the `max`-operator. In contrast, the second two methods separate evaluation and improvement of the policy. In this respect, value iteration can be considered as off-policy because it aims at directly estimating V^* whereas policy iteration estimates values for the current policy and is on-policy. However, in the model-based setting the distinction is only superficial, because instead of samples that can be influenced by an on-policy distribution, a model is available such that the distribution over states and rewards is known.

2.6.3 Efficient Exploration and Value Updating

The methods in the previous section have shown that both prediction and control can be learned using samples from interaction with the environment, without having access to a model of the MDP. One problem with these methods is that they often need a large number of experiences to converge. In this section we describe a number of extensions used to speed up learning. One direction for improvement lies in the exploration. One can – in principle – use MC sampling until one knows everything about the MDP but

this simply takes too long. Using more information enables more focused exploration procedures to generate experience more efficiently. Another direction is to put more efforts in using the experience for updating multiple values of the value function on each step. Improving exploration generates *better samples*, whereas improving updates will squeeze *more information* from samples.

Efficient Exploration. We have already encountered ϵ -greedy and *Boltzmann* exploration. Although commonly used, these are relatively simple *undirected* exploration methods. They are mainly driven by *randomness*. In addition, they are *stateless*, i.e. the exploration is driven without knowing which areas of the state space have been explored so far. A large class of *directed* methods for exploration have been proposed in the literature that use additional information about the learning process. The focus of these methods is to do more uniform exploration of the state space and to balance the relative benefits of discovering new information relative to exploiting current knowledge. Most methods use or learn a model of the MDP in parallel with RL. In addition they learn an *exploration value function*. Several options for directed exploration are available. One distinction between methods is whether to work *locally* (e.g. exploration of individual state-action pairs) or *globally* by considering information about parts or the complete state-space when making a decision to explore. Furthermore, there are several other classes of exploration algorithms.

Counter-based or *recency-based* methods keep records of how often, or how long ago, a state-action pair has been visited. *Error-based* methods, of which *prioritized sweeping* (Moore and Atkeson, 1993) is one example, use an exploration bonus based on the error in the value of states. Other methods base exploration on the *uncertainty* about the value of a state, or the *confidence* about the state's current value. They decide whether to explore by calculating the probability that an explorative action will discover a larger reward than already found. The *interval estimation* (IE) method by Kaelbling (1993b) is an example of this kind of methods. IE uses a statistical model to measure the degree of uncertainty of each $Q(s, a)$ -value. An upper bound can be calculated on the likely value of each Q -value, and the action with the highest upper bound is taken. If the action taken happens to be a poor choice, the upper bound will be decreased when the statistical model is updated. Good actions will continue to have a high upper bound and will be chosen often. In contrast to counter- and recency-based exploration, IE is concerned with *action exploration* and not with *state space* exploration. Wiering (1999) (see also Wiering and Schmidhuber, 1998a) introduced an extension to model-based RL in the *model-based interval estimation* algorithm in which the same idea is used for estimates of transition probabilities.

Another recent method that deals explicitly with the exploration-exploitation trade-off is the E^3 method by Kearns and Singh (1998). E^3 stands for *explicit exploration and exploitation*. It learns by updating a model of the environment by collecting statistics. The state space is divided into *known* and *unknown* parts. On every step a decision is made whether the known part contains sufficient opportunities for getting rewards or whether the unknown part should be explored to obtain possibly more reward. An important aspect of this algorithm is that it was the first general near-optimal (tabular) RL algorithm with provable bounds on computation time. The approach was extended by Brafman and Tennenholtz (2002) into the more general algorithm R-MAX. It too provides a polynomial bound on computation time for reaching near-optimal policies. As a last example, Ratitch (2005) presents an approach for efficient, directed exploration based on more sophisticated characteristics of the MDP such as an *entropy* measure over state transitions. An

interesting feature of this approach is that these characteristics can be computed *before* learning and be used in combination with other exploration methods, thereby improving their behavior.

For a more detailed coverage of exploration strategies we refer the reader to (Ratitch, 2005) and (Wiering, 1999).

Guidance and Shaping. Exploration methods can be used to speed up learning and focus attention to relevant areas in the state space. The exploration methods mainly use statistics derived from the problem before or during learning. However, sometimes more information is available that can be used to *guide* the learner. For example, if a reasonable policy for a domain is available, it can be used to generate more useful learning samples than (random) exploration could do. In fact, humans are usually very bad in specifying optimal policies, but considerably good at specifying reasonable ones⁸.

The work in *behavioral cloning* (Bain and Sammut, 1995) takes an extreme point on the guidance spectrum in that the goal is to *replicate* example behavior from expert traces, i.e. to *clone* this *behavior*. This type of guidance moves learning more in the direction of *supervised* learning. Another way to help the agent is by *shaping* (Mataric, 1994; Dorigo and Colombetti, 1997; Ng *et al.*, 1999). Shaping pushes the reward closer to the subgoals of behavior, and thus encourages the agent to incrementally improve its behavior by searching the policy space more effectively. This is also related to the general issue of giving rewards to appropriate subgoals, and the gradual increase in difficulty of tasks. The agent can be trained on increasingly more difficult problems, which can also be considered as a form of guidance.

Various other mechanisms can be used to provide guidance to RL algorithms, such as decompositions (see Dixon *et al.*, 2000, and further hierarchical decompositions in the next chapter), heuristic rules for better exploration (Främling, 2005) and various types of *transfer* in which knowledge learned in one problem is transferred to other, related problems (e.g. see Konidaris, 2006, and further in Chapters 4 and 7).

Eligibility Traces. In MC methods, the updates are based on the entire sequence of observed rewards until the end of an episode. In TD methods, the estimates are based on the samples of immediate rewards and the next states. An intermediate approach is to use the *n-step-truncated-return* $R_t^{(n)}$, obtained from a whole sequence of returns:

$$R_t^{(n)} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^n V_t(s_{t+n})$$

With this, one can go to the approach of computing the updates of values based on several *n*-step returns. The family of $\text{TD}(\lambda)$, with $0 \leq \lambda \leq 1$, combines *n*-step returns weighted proportionally to λ^{n-1} .

The problem with this is that we would have to wait indefinitely to compute $R_t^{(\infty)}$. This view is useful for theoretical analysis and understanding of *n*-step backups. It is called the *forward view of the TD(λ) algorithm*. However, the usual way to implement this kind of updates is called the *backward view of the TD(λ) algorithm* and is done by using *eligibility traces*, which is an incremental implementation of the same idea.

⁸Quote taken from the invited talk by Leslie Kaelbling at the European Workshop on Reinforcement Learning (EWRL), in Utrecht in 2001.

Eligibility traces are a way to perform n -step backups in an elegant way. For each state $s \in S$, an eligibility $e_t(s)$ is kept in memory. They are initialized at 0 and incremented every time according to:

$$e_t(s) = \begin{cases} \gamma \lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma \lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases}$$

where λ is the *trace decay parameter*. The trace for each state is increased every time that state is visited and decreases exponentially otherwise. Now δ_t is the *temporal difference* error at stage t :

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

On every step, all states are updated in proportion to their eligibility traces as in:

$$V(s) = V(s) + \alpha \delta_t e_t(s)$$

The forward and backward view on eligibility traces can be proved equivalent (Sutton and Barto, 1998). For $\lambda = 1$, TD(λ) is essentially the same as MC, because it considers the complete return, and for $\lambda = 0$, TD(λ) uses just the immediate return as in all one-step RL algorithms. Eligibility traces are a general mechanism to learn from n -step returns. They can be combined with all of the model-free methods we have described in the previous section. Watkins (1989) combined Q -learning with eligibility traces in the $Q(\lambda)$ -algorithm. Peng and Williams (1996) proposed a similar algorithm, and Wiering and Schmidhuber (1998b) and Reynolds (2002) both proposed efficient versions of $Q(\lambda)$. The problem with combining eligibility traces with learning *control* is that special care has to be taken in case of *exploratory* actions, which can break the intended meaning of the n -step return for the current policy that is followed. In Watkins (1989)'s version, eligibility traces are reset every time an exploratory action is taken. Peng and Williams (1996)'s version is, in that respect more efficient, in that traces do not have to be set to zero every time. SARSA(λ) (Sutton and Barto, 1998) is more safe in this respect, because action selection is on-policy. Another recent on-policy learning algorithm is the QV(λ) algorithm by Wiering (2005). In QV(λ)-learning two value functions are learned; TD(λ) is used for learning a state value function V and one-step Q -learning is used for learning a state-action value function, based on V .

Learning and Using a Model: Learning and Planning. Even though RL methods can function without a model of the MDP, such a model can be useful to speed up learning, or bias exploration. A learned model can also be useful to do more efficient value *updating*. A general guideline is that when experience is costly, it pays off to learn a model. In RL model-learning is usually targeted at the specific learning task defined by the MDP, i.e. determined by the rewards and the goal. In general, learning a model is most often useful because it gives knowledge about the *dynamics* of the environment, such that it can be used for other tasks too (see Drescher, 1991, for extensive elaboration on this point).

The DYNA architecture (Sutton, 1990, 1991b,a; Sutton and Barto, 1998) is a simple way to use the model to amplify experiences. Algorithm 5 shows DYNA- Q which combines Q -learning with planning. In a continuous loop, Q -learning is interleaved with series of extra updates using a model that is constantly updated too. DYNA needs less interactions with the environment, because it *replays* experience to do more value updates.

A related method that makes more use of experience using a learned model is *prioritized sweeping* (PS) (Moore and Atkeson, 1993). Instead of selecting states to be updated

Algorithm 5 DYNQ-Q (Sutton and Barto, 1998).

Require: initialize Q and Model arbitrarily

```

1: repeat
2:    $s \in S$  is the start state
3:    $a := \epsilon$ -greedy( $s, Q$ )
4:   update  $Q$ 
5:   update Model
6:   for  $i := 1$  to  $n$  do
7:      $s :=$  randomly selected observed state
8:      $a :=$  random, previously selected action from  $s$ 
9:     update  $Q$  using the model
10: until sufficient performance
    
```

randomly (as in DYNQ), PS prioritizes updates based on their change in values. Once a state is updated, the PS algorithm considers all states that can reach that state, by looking at the transition model, and sees whether these states will have to be updated as well. The order of the updates is determined by the size of the value updates. The general mechanism can be summarized as follows. In each step **i**) one remembers the old value of the current state, **ii**) one updates the state value with a full backup using the learned model, **iii**) one sets the priority of the current state to 0, **iv**) one computes the change δ in value as the result of the backup, **v**) one uses this difference to modify *predecessors* of the current state (determined by the model); all states leading to the current state get a priority update of $\delta \times T$. The number of value backups is a parameter to be set in the algorithm. Overall, PS focuses the backups to where they are expected to most quickly reduce the error. More information about PS will be given in Chapter 5, where it is used in relational domains. Another example of using planning in model-based RL is (Wiering, 2002)

2.7. Beyond the Markov Assumption

Throughout the chapter we have described models and algorithms that rely heavily on the so-called Markov assumption, i.e. the fact that a given state s contains all the necessary information to make an optimal decision. Evidently, some amount of work is needed to actually compute that optimal decision for s , for example using iterative value update procedures based on the Bellman equations. But once that is done, the result is a mapping from any state s to an optimal action a , and this mapping is independent of how s was reached; the only thing we need to know is that we are in s . In the following chapters we will dissect the atomic states and find variations on how states look like, for example using binary features, or relational facts. Yet, in all these cases, the basic assumption is that states originate from what an agent can perceive from its environment through *sensors* (whatever they may be) and that when given such a sensor input, the agent can make an optimal decision.

But, what if the states do not contain enough information for an optimal decision? For example, the agent's sensors might be faulty such that they give a wrong impression of how the world looks like. Or maybe there is information that the agent simply cannot see or is not allowed to see. Consider the robot in Figure 2.4a. The robot can move through the corridor (positions 1–5). There are two rooms, labeled X and \$, and the locations 2

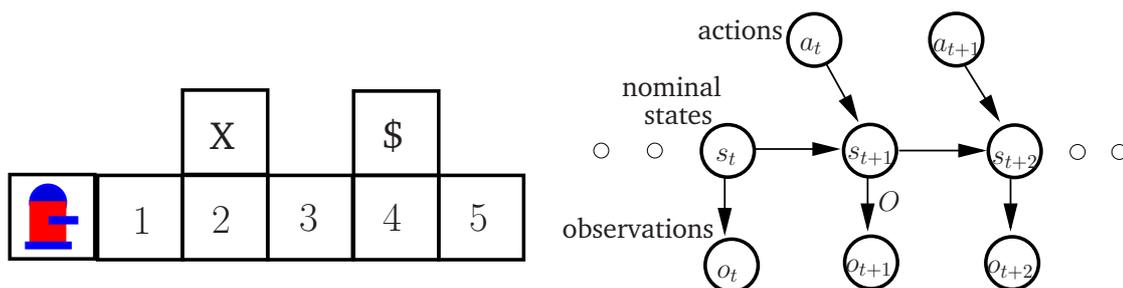


Figure 2.4: a) A typical example of a partially observable grid world. b) The dynamics of a partially observable Markov decision process (POMDP).

and 4 contain doors to these rooms. The robot's sensors can only⁹ sense the presence of walls and doors in four directions. That means that both positions 2 and 4 look exactly the same for the robot. Now consider a reward function that gives a large negative reward to state X and a large positive reward to state \$. How can the robot learn to move to \$ instead of X in an MDP setting? The answer is, it cannot. Because it would have to learn a value function or a policy that maps the same state to two different values or actions. But, let us assume that the robot always starts moving from the position in the picture. In that case, what is needed is that the robot remembers whether he has already seen a door when moving to the right. If it has not, then it has to continue moving. If it has, then it is already past the door to the room with X, and it can enter the second door, which leads to \$. Such situations are a kind of *road sign problem* (Rylatt and Czarnecki, 2000), which is a class of delayed response tasks in which an agent's correct turning direction at a T -junction is dependent on a stimulus (i.e. the road sign) it has encountered earlier. Storing such events can be done by extending the state representation, with the goal of making the difference between states 2 and 4 explicit.

A general way to model problems where the current state is not informative enough for optimal decisions, is to define a k -order Markov model. Definition 2.2.1 is about 1-Markov systems, where the optimal action can be computed using only the most recent single state, and forms the basis for MDPs. In a k -order Markov, the k most recent states are important for this decision. The 1-Markov, or more generally just Markov, model is the core topic of this book. In the next two paragraphs we briefly discuss two of the research directions that have gone beyond the Markov assumption. We do this for several reasons. One is that the k -Markov case is a generalization of the Markov case, and therefore many of the models and techniques in the coming chapters are being employed for this too. A second reason is that in the next chapter we describe abstraction and generalization techniques in the MDP context, which can generate new models which are not Markov anymore and thus similar problems arise in this setting. One additional reason is that – although current relational RL is almost entirely occupied with MDPs, presumably in the near future, partially observable, relational MDPs will be studied.

⁹One might also consider the case where the robot can only sense that there is some solid object (which may be a door or a wall) in its surroundings. In that case the problem gets even harder, because now e.g. position 3 looks exactly like positions 2 and 4.

2.7.1 Partially Observable Markov Decision Processes

In general, most realistic problems display various forms of *partial observability*. The most basic strategy is to ignore this aspect and use the MDP as a model. This works for limited cases, but in general one has to deal explicitly with this problem by extending the MDP to a *partially observable Markov decision process* (POMDP) (Kaelbling *et al.*, 1998). A POMDP is an MDP in which the agent is unable to observe the true current state. Instead, it makes an observation based on the action and resulting state.

DEFINITION 2.7.1 ▶ A **partially observable Markov decision process** is a tuple $\langle S, A, T, R, O, \Omega \rangle$ such that $\langle S, A, T, R \rangle$ forms an MDP as defined as in Definition 2.2.1, O is a set of observations and Ω is the observation function defined as $\Omega : S \times A \times O \rightarrow [0, 1]$ defining a probability distribution over observations for each state-action pair.

Figure 2.4b depicts the dependencies between the nominal states, actions and the observations. POMDPs are capable of modeling more complex domains, at the expense of computationally much more demanding algorithms for computing optimal value functions and policies. POMDPs generalize MDPs by allowing incomplete information about the state. A *belief state* b is a probability distribution over S , stating for each state $s \in S$ the probability $b(s)$ that s is the real, current state. In addition to the sets of states and actions, the initial and goal situations, and the transition and reward functions, a POMDP involves *prior beliefs* in the form of a probability distribution over S and a *sensor model* in the form of O of possible observations and probabilities $P(o | s, a)$ of observing $o \in O$ in state s after doing the action a . While the agent cannot predict the effects of actions on the states, it can predict the effect of actions on the belief state. The new belief state b_a , representing the result of applying action a in the belief state b can be computed by conditioning on the observation:

$$b_a(s) = \sum_{s' \in S} P(s | s', a) b(s') \quad (2.20)$$

$$b_a(o) = \sum_{s \in S} P(o | s, a) b_a(s) \quad (2.21)$$

$$b_a^o(s) = \frac{P(o | s, a) b_a(s)}{b_a(o)}, \text{ if } b_a(o) \neq 0 \quad (2.22)$$

where $b_a^o(s)$ represents the probability that results from having done action a in belief state b and having observed o afterwards. Thus, in contrast with MDPs, where updating the current (belief) state is simple, updating POMDP belief states require considerably more work.

Model-based solution techniques for POMDPs can make use of the following generalization of the Bellman backup operator:

$$V^{k+1}(b) = \mathbf{B}^*(V^k(b)) = \max_{a \in A} \left\{ \sum_{o \in O} r(b, a, o) + \gamma P(o | b, a) V(b_o^a) \right\} \quad (2.23)$$

In this way, DP backups work over a so-called *belief state* MDP, which is continuous, but it has been shown that they have piecewise and convex value functions and can be solved by value iteration or policy iteration. In Section 6.1.5.4 we will return to this topic. For many problems, obtaining exact solutions becomes computationally intractable, and for

that reason most recent techniques use various forms of approximation (see Aberdeen, 2003; Pineau *et al.*, 2006; Smith and Simmons, 2005; Spaan, 2006, for recent overviews). Approximations employ similar methods as those that will be described in the next chapter for MDPs, such as *factored representations* and *policy search*, and they can be applied in contexts with abstraction, generalization, *hierarchical decompositions* and *function approximation*.

When the model of the POMDP is not available, things become more complex. In that case, the agent has to learn from observations alone. The main problem (see also the example in Figure 2.4) is that it is often not known beforehand how many past observations are needed for optimal decisions. The simplest case is just to forget about partial observability and use a model-free learning algorithm such as Q -learning, ensuring that a stochastic (reactive and memory-less) policy is used. However, a more general solution to the problem requires taking into account some form of *history* of past observations. These can range from simple memory devices (e.g. see Lanzi, 2000) to various forms of *recurrent neural networks* (e.g. see Lin, 1992; Großmann, 2001; Bakker, 2004, and see also Section 3.6.2.2). Alternative approaches are *hierarchical decompositions* (e.g. see Wiering and Schmidhuber, 1997, and also Section 3.8) and *decision-tree algorithms* (e.g. see McCallum, 1996, and also Section 3.6.2.3). As value functions for POMDPs can be complex, *evolutionary algorithms* have also been used as an alternative to find approximate policies (Schmidhuber, 2000; Moriarty *et al.*, 1999).

For more information about POMDPs, and exact and approximate algorithms for solving them, there are many recent overviews (Kaelbling *et al.*, 1998; Murphy, 2000; Russell and Norvig, 2003; Aberdeen, 2003; Spaan, 2006).

Predictive Representation of State. Methods that use k -Markov models attempt to identify state by remembering what has happened in the past. In the context of POMDPs we have briefly described a second method in which belief states identify state as a distribution over (postulated) nominal states. A third type of models for dynamical systems is known as *predictive representations* (Littman *et al.*, 2001; Singh *et al.*, 2004), which identify state by predicting what will happen in the *future* instead of building it from past observations. Conceptually, the difference can be illustrated by a difference between questions such as "is there a car behind me?" and "what would I see if I looked in my rear mirror?", and the difference between "is my partner home?" and "if I called home, what would be the probability that my partner answers my call?".

A *predictive state representation* (PSR) is a compact and complete description of a dynamic system, and it represents the belief about the state of the world as a set of probability distributions over *tests*. A test is a sequence of actions and observations that can be executed at a given time. A test is executed if we execute each of its specific actions in order. It succeeds if it is executed and the observations produced by the dynamical system match those specified in the test. So, for example, if a dynamical system has actions $\{a_1, a_2, a_3\}$ and observations $\{o_1, o_2, o_3\}$ then it might produce the action-observation sequence

$$a_1 o_2 a_1 o_3 a_2 o_3 a_3 o_2 a_1, \dots$$

starting at $t = 0$. Now we could say that at time 0 the test $a_1 o_2 a_1 o_3$ is successfully executed, that at time 1 the test $a_1 o_2 a_2 o_3$ was executed unsuccessfully and that the test $a_1 o_1 a_1 o_1$ was never executed.

PSRs have been shown to be superior to POMDPs in terms of the ability to efficiently model problems and provide solutions. PSRs are designed to predict all possible future test sequences of instances using some projection function over the given probability of observing a set of core tests. Once learned, a PSR implicitly defines states as equivalence classes that have the same core test probabilities, and provides a basis for RL algorithms. Most current PSRs use linear combinations of core tests, but the field is rapidly exploring new representations and algorithms, especially for learning and discovering PSRs. Examples are TD-Networks (Sutton and Tanner, 2005) which generalize temporal-difference learning to networks of interrelated predictions, and *schema learning* (Holmes and Isbell jr., 2004) (an extension of the schema mechanism by Drescher (1991)), a constructivist approach that incrementally builds up predictors of action effects. PSRs are being extended with the same kind of abstraction mechanisms we will distinguish in the next chapter, for example *temporal* abstraction (Sutton *et al.*, 2005). Much more work and ideas are needed to find automatic ways of *discovering* PSRs and TD-networks (e.g. as in PIAGET-3, but see Makino and Takagi, 2008, for some recent work in this direction).

2.8. Discussion

Markov decision processes provide a formal framework for capturing a large class of sequential decision making tasks. They come with a number of essential assumptions. The first is that the system to be controlled, i.e. modeled as an MDP, consists of a set of distinguished *states*. A state is assumed to form an adequate description of the current state of the system that provides enough information for optimal behavior in that particular state. A second assumption is that there is a number of distinguished *actions* that can be executed in each state. The result of action execution in a state s is a transition to another state, governed by a fixed probability distribution that specifies the probability of making the transition to another state s' . It is questionable whether in any real application, a complete transition model is known beforehand. But, as we have seen, models can be learned, or value function and policy learning can be done even without the model, by sampling the MDP. Transition probabilities are only dependent on the current state and the action that was chosen. This is called the Markov property and it is the most important aspect of MDPs. It enables computation of optimal policies that map states into actions. A third assumption is that an MDP comes with a *reward function* that assigns numeric rewards to each transition, or each state. The reward function is supposed to reflect the task that is modeled by a given MDP. If the reward function is perfectly aligned with the task, a policy that will gather the most reward when executing its actions in the MDP will be optimal, and be best at performing the task. Rewards are usually assumed to come from *outside* the learning agent, however some approaches have tried to capture cases where the agent is *intrinsically* motivated (e.g. see Singh *et al.*, 2005), or possesses a value system and is *self-supervising* (e.g. see Pfeifer and Scheier, 1999, Ch. 14).

The assumptions about the states and actions are very strong for most environments. The fact that there is a certain state of affairs at each decision moment may be hard to ensure. In a concrete application, most likely complete information is not available because some aspects may be hidden, or some aspects cannot be reliably observed through the available sensors. This is why much of current RL research goes beyond the Markov assumption and tries to learn policies by assuming there is no such thing as *the* state, but

there may be probabilistic information about the real state obtained from observations (as in POMDPs). Predictive representations of state go even further by assuming that there is no state, only observations. In this book we limit our discussion to MDPs, because this is the type of models that is targeted in MDPs that use *first-order* knowledge representation, the main topic of this book. However, there is no doubt that in the near future, the strong assumptions of MDPs will have to be loosened in this context too.

In this chapter we have defined the MDP setting. We have described two branches of algorithms – model-based and model-free – and for each of them several efficient extensions. In both branches, the extensions dealt with more efficient explorations of the state space and with more efficient updates. The first type of extension was able to make the relevant parts of the environment smaller such that learning could be faster. The second type of extension tried to get more information from individual samples such that updates of value functions and models are more effective, which decreases the number of learning samples needed for computing optimal policies. Although these methods and especially their extensions enable to scale up to larger problems, the discrete, finite model case of this chapter will not scale up to arbitrarily large, real-world problems. For this, we need more *compact* and *structured* representations, which will be the topic of the next chapters. There we will find many different formalisms for more compact *representations* and algorithms that make use of these representations for more efficient learning. However, all these methods will be based on the models and algorithms defined in this chapter.

Generalization and Abstraction in Markov Decision Processes

Abstraction is one of most important tools of the computer scientist. In fact, the design and implementation of complex systems cannot be done without it. In general, abstraction is used to simplify elements of complex systems such that complexity is lowered without decreasing performance significantly. In artificial intelligence and machine learning, its main goal is to lower the complexity of reasoning and learning without losing the ability to deal effectively with the task. In this chapter, we focus on abstraction mechanisms used in the context of MDPs. The learning algorithms from the previous chapter do not scale up well to larger state spaces. Various abstractions are proposed in the literature that make use of the inherent structure in MDPs such that even very large problems can be solved. Structure can be found in symmetries and equivalence classes in state and action spaces, in task hierarchies and also in regularities in transition models and reward functions. Making use of abstractions renders a need to deal with structural induction and deduction in parallel with value and policy learning algorithms. The main purpose of this chapter is to highlight the principles of abstraction in MDPs and the interplay between abstractions and RL algorithms. We introduce PIAGET as an extension of GPI in the face of abstraction. Furthermore, we distinguish five types of abstraction for solution of MDPs and the same underlying principles can be found in the following chapters covering relational representations.

THE MARKOV DECISION PROCESS (MDP) framework has become a *de facto* standard method for learning sequential decision problems in which a performance metric is available to optimize decision making in the context of uncertainty. It provides a general modeling framework for many interesting tasks such as (probabilistic) planning, game-playing and goal-seeking behavior. *In principle*, all (fully-observable) tasks in which there is a (numerical) performance measure available that can evaluate behavior, can be solved in the MDP framework. The previous chapter has introduced MDPs and solution algorithms based on *explicit* state and action representations.

However, computing solutions for systems at the most fine-grained level is often inconvenient and inefficient. If we analyze humans learning and reasoning about complicated tasks, we see that they use *abstraction* and *generalization* techniques that enable them to see and use the *inherent structure* present in many tasks (see Baum, 2004). For example, consider a board game such as CHESS, GO, or CHECKERS. When explaining the game and

	O	
X		O
*		X

	O	X
O		
	X	*

X		*
O		X
	O	

*	X	
		O
X	O	

Figure 3.1: Symmetries in TIC-TAC-TOE. An optimal move for X is denoted $*$ in each board position. All four positions with their corresponding optimal action are equivalent when taking rotational symmetries into account. The actions ($*$) in these positions describe create so-called fork positions. The opponent O cannot block both lines of X 's, such that X will win in the next move.

its purpose to new players, we do not go into explaining each board position individually and which actions are possible in each situation, simply because most games have enormous state spaces. In fact, the state space of the simple game of TIC-TAC-TOE contains already (roughly) 6000 states. Instead, the game is explained in terms of general *rules*, *properties* of states and *abstract goals*. For TIC-TAC-TOE one would explain that there are two types of symbols (one for each player), that there are 9 squares, that each player has to put its symbol on an *empty* square and that there are *lines* on the board such that when one player fills one line entirely with his or her symbols, he or she wins the game (i.e. gets a reward +1). A suitable *policy* for playing the game makes use of the lines on the board, and *patterns* of symbols on the board. TIC-TAC-TOE contains a considerable amount of *symmetry* such that knowledge about an optimal action in one position can be generalized to other positions that are *similar* with respect to these symmetries, see Figure 3.1. Humans can master the game fairly quickly by making use of the *patterns* on the board and symmetries between board positions. There is no need to see all possible positions to compute an optimal strategy, provided that one makes use of abstraction and generalization.

The field of *artificial intelligence* (AI) (Russell and Norvig, 2003; Görtz *et al.*, 2003; Luger, 2002) shows a wide variety of abstraction and generalization techniques that can be used to highlight *structure* in problem domains. *Knowledge representation* (KR) (Markman, 1999; Sowa, 1999; Brachman and Levesque, 2004) formalisms play an important role in *representing* and *reasoning about* problems in *compact*, *comprehensible* and *efficient* ways. The field of *machine learning* (Langley, 1996; Mitchell, 1997; Alpaydin, 2004) provides many ways for the *induction* of *compact hypotheses* that *generalize* over problem instances. All these approaches can be used in the context of MDPs. Many types of *abstraction* can be used for compact representations of states, actions, transition and reward functions, policies, and task structures, such that representing and learning can be performed on conceptually higher levels than that of individual states and actions (see for example Bertsekas and Tsitsiklis, 1996; Sutton, 1997; Sutton and Barto, 1998; Boutilier, 1999; Boutilier *et al.*, 1999). Furthermore, *generalization techniques* such as *function approximators* can *learn* compact mappings from states to values. The algorithms in the previous chapter do not *scale up* to arbitrarily large problems, due to the sheer *size* of state spaces, which is even *infinite* for *continuous* state spaces.

In this chapter we will discuss various abstraction, generalization and KR techniques used in the context of MDPs. We focus on methods that use MDPs and RL algorithms as their *main* components. It is unlikely that it is possible to develop a general-purpose algorithm that can approximate solutions to arbitrary sequential decision problems. Instead there will be more likely a whole range of algorithms that exploit different types of

structures to approximate solutions. We structure our exposition of the literature along various dimensions of *what* is being abstracted (e.g. value functions or policies) and *how* they are abstracted. We furthermore distinguish between methods using *fixed* abstractions and methods using *adaptive* abstractions.

Goals and Outline of this Chapter. We have several distinct goals in this chapter. First, we want to show how (propositional) abstraction and generalization can be employed in the MDP framework. But, at the same time, we will impose structure on the various ways this can be done. For doing this, we introduce the PIAGET principle, as an extension of generalized policy iteration where we make the use of abstraction and generalization explicit. Second, in parallel with this principle, we distinguish five main types of abstraction in MDPs, and for each of these types, we describe their main characteristics. These descriptions will survey a large, representative part of the literature. On the one hand, this is because we want illustrate our conceptual framework with many examples. On the other hand, we want to show many types of *rich representations* that have been used in the literature. More specifically we focus on those types of abstraction and those algorithms that can – and have – be upgraded to first-order domains starting from Chapter 4.

We start this chapter with shifting the focus from an *algorithmic* to a *representational* point of view on RL in Section 3.1 and furthermore a description of the general concept of *abstraction* in Section 3.2. This involves properties and theories, as well as motivations and advantages. After that we turn to abstraction mechanisms in a more narrow context, in the MDP framework in Section 3.3, where we distinguish 5 different types of abstraction. We will introduce the PIAGET-principle as a general mechanism behind MDP solution techniques in the face of abstraction. After this, we cover the 5 types of abstraction in the subsequent sections. In Section 3.4 we describe methods that try to reduce the size of the model by aggregating model components into larger ones, sometimes directly from the model description. After that, in Section 3.6, we turn to *value function approximation* methods that use function approximators (e.g. *regression engines*) as a replacement for tables representing value functions. Another class of abstraction mechanisms, so-called *policy search* approaches, is described in Section 3.7, in which abstract versions of policies are learned directly, without explicitly representing and learning value functions. The area of *factored representations* for MDPs is described in Section 3.5 and this section partially deals with *structured* versions of value and policy iteration. In Section 3.8 we describe *hierarchical* approaches to RL, in which decompositions in terms of *task structures* are learned and used. We describe briefly some examples of abstraction and generalization in an RL application in the area of *automatic fingerprint recognition* in Section 3.9 and we conclude this chapter with a discussion in Section 3.10.

3.1. From Algorithmic to Representational

Chapter 2 has described foundational approaches to *value-based* learning for MDPs. One of the founders of modern RL, Sutton (1999), describes briefly the core directions in the short history of the field so far. He distinguishes between RL *past*, RL *present* and RL *future*. These directions are interesting because they form the basis for a number of chapters in this book.

The RL *past* encompasses the period until approximately 1985 in which the idea of *trial-and-error* learning was developed. This period emphasized the use of an active, exploring

agent and developed the key insight of using a scalar reward signal to specify the *goal* of the agent, termed the *reward hypothesis*. The methods usually only learned policies and were generally incapable of dealing effectively with delayed rewards.

The RL *present* was the period in which *value functions* were formalized. Value functions are at the heart of RL and virtually all methods focus on approximations of value functions in order to compute (optimal) policies. The *value function hypothesis* says that approximation of value functions is the dominant purpose of intelligence. Chapter 2 is about RL *present*, in that it describes value functions and *temporal difference* methods for learning them, and additionally *efficient extensions* for faster learning and larger problems.

At this moment, we are in the RL *future*. Sutton made predictions about the direction of this period, and it is this direction that we describe in this chapter. In Sutton's words,

*"Just as RL present took a step away from the ultimate goal of reward to focus on value functions, so RL future may take a further step away to focus on the structures that enable value function estimation [...] In psychology, the idea of a developing mind actively creating its representations of the world is called **constructivism**. My prediction is that for the next tens of years RL will be focused on constructivism."*

Constructivism as a mechanism has been studied in fields such as AI (Drescher, 1991; Shultz, 2003), *computational neuro-science* (Elman *et al.*, 1996; Quartz and Sejnowski, 1997; Quartz, 1999), *developmental psychology* (Piaget, 1950; Thornton, 2002) and *machine learning* (Utgoff and Precup, 1997, 1998; Thornton, 1999, 2000; Utgoff and Stracuzzi, 2002). The underlying concept is that a developing entity is supplied with no, or very limited, *bias* or *a priori knowledge* and has to make sense of the world in an autonomous fashion. In the context of RL and this book, constructivism is defined more narrowly in that it aims at automating the process of learning abstractions of MDPs, value functions and policies. This constructivist direction is the topic of this and subsequent chapters.

So far, we have only witnessed a some years of what Sutton calls RL *future*, but during this period and already before, a large number of methods have been proposed that deal with these *structures enabling value function estimation* Sutton talks about. Various choices for representations of states, actions, world dynamics, task structures and many more, have been described and validated. The common goal of these methods is to scale up to larger problems, by using better representational formalisms and by using *a priori* knowledge about the domain or the task¹. *Rich representational formalisms* can be used to construct *compact representations* such that sequential decision making can be learned on higher *abstraction levels*. The ultimate goal is to have *complete* constructivism, i.e. that all structures are induced autonomously by the agent². However, we will see that in the

¹In RL one often encounters the phrase that "we have to give up *tabula rasa*" learning (see e.g. Kaelbling *et al.*, 1996, p. 268). *Tabula rasa* is Latin for blank slate, and traditional, classical RL algorithms are often criticized by starting from scratch, without taking any prior knowledge into account.

²However, modern theories concerning cognitive development suggest that for example newborns have much of what they need to know in order to understand objects is already pre-programmed at birth. The reason for this is that the *perceptual systems* are very sophisticated in the newborn, but their ability to *use* the information it provides, is very limited. (Thornton, 2002). In the context of this chapter, if we assume a functioning perceptual system providing the right *features* for some problem, we might still go for full constructivism from there, i.e. the development of structures that *use* this information.

current state-of-the-art, most methods still rely on a narrowly defined context and considerable *bias* supplied by the designer of the system. All recent directions in RL that go beyond table-based representations and algorithms can be taken under the general name of *rich representations*³. Rich representations open up many possibilities for solving larger and more complex problems, although they also introduce new challenges for algorithms to deal efficiently with these new representational capabilities. In the following, let us first outline the mechanisms and problems of simple, table-based representations.

3.1.1 Algorithmic Aspects

The previous chapter described several classes of algorithms for computing optimal value functions and policies. A crucial assumption in these algorithms is that all computation is done *at the level of individual states and actions*. So-called *backup tables* keep the information about state values, state-action values, transition probabilities and reward distributions for individual elements (e.g. states and actions) of the MDP. Most algorithms make heavily use of Bellman equations for backing up values for states and state-action pairs. Backing up values can be performed *fully* by using a known model, or they can be performed more *locally* based on sampled traces. In both cases, convergence can be assured under mild assumptions. The underlying structure in all these algorithms is the *generalized policy iteration* (GPI) loop (see Figure 2.3) in which two separate computational processes interact. The *policy evaluation* step computes a value function for a given (fixed) policy and the *policy improvement* step improves the current policy based on the information computed in the evaluation step. Many possibilities exist for both steps to make algorithms more efficient (see Sections 2.5.2 and 2.6.3).

Under the assumption that all information is stored and computed at the individual level of states and actions, the *increased efficiency* of the solution algorithms for MDPs is obtained along two dimensions. The first dimension is the *order* in which the updates are performed. In model-based algorithms this order can be changed by performing sweeps of value updates in various orders (e.g. modified policy iteration), by more fine-grained interleaving of value updates and policy updates (e.g. value iteration) and by making use of search (e.g. RTDP). In model-free algorithms, the order is determined by the experienced traces, which is influenced by the exploration strategy. The second dimension is the *selection* of which values get updated. In model-free algorithms the selection coincides with sampled traces through the state space, and this is determined by the current policy and exploration strategy. Selection in model-based algorithms can be influenced using (heuristic) search or by neglecting unreachable, or less relevant, parts of the state space. Methods such as DYNA and *prioritized sweeping* use the transition model to determine which values get updated. In fact, this introduces the concept of *complex backups*, in which one sampled experience is used to update *multiple* values.

Summarizing, the previous chapter has shown a number of *algorithms* that can solve MDPs (optimally), and in addition a variety of *efficient extensions* of these algorithms. The efficient extensions speed up computation and convergence of the algorithms and this enables scaling up to larger problems. Still, the assumption that all information is stored

³The growing interest in using rich representations in reinforcement learning is also supported by recent events such as the *Relational Reinforcement Learning* workshop, <http://eecs.oregonstate.edu/research/rrl/> at ICML'04 and the *Rich Representations for Reinforcement Learning* workshop <http://www.cs.waikato.ac.nz/~kurd/rrfrl/> at ICML'05.

and computed at the level of individual states and actions, is unreasonable – and even unwanted – for most realistic problems. Seemingly small problems can already result in huge state spaces which are too large to store in tables. Consider as an example the game of AWARI. This ancient, African, 2-player game has its origins in Africa and is played using only a set 12 of holes in the ground and 48 stones. The stones are put in the holes and can be moved around by the players following a few simple rules⁴. The goal of the game is to gain a majority of stones. With only these 48 stones and 12 holes, the game’s state space contains 889.063.398.406 states and a full transition matrix contains about 7.9×10^{23} entries for each action (although it is very sparse). Estimating these probabilities is an additional problem, because they depend on the other player’s strategy. These numbers are simply too large for any modern computer’s memory. Applying RL to this problem would be virtually impossible, just for storage reasons alone⁵. Even if storing these huge matrices, and obtaining a correct transition matrix, is not a problem, the complexity of learning an optimal policy is. The algorithms for computing value functions and policies need large numbers of sweeps through the state space (e.g. for model-based algorithms) and large numbers of sampled traces (e.g. for model-free algorithms) in order to converge.

3.1.2 Fundamental Problems of Huge State Spaces

Basically, there are three fundamental problems with computing optimal value functions and policies for MDPs with large state spaces. The first is the *storage problem*: seemingly simple problems can quickly induce huge tables that have to be stored, retrieved and updated. The second problem is the *learning problem*: convergence of algorithms depends on large numbers of updates of an already large number of elements. The third problem is the *needle-in-a-haystack problem*: when state spaces are huge, getting necessary feedback for learning can be hard to find. For example, in environments containing only one designated *goal* area where non-zero reward can be obtained, stumbling upon this area by the initial (random) exploration strategy is virtually impossible. Finding this area using random exploration in a huge state space is similar to looking for a needle in a haystack. The use of guidance, directed exploration, and other strategies can enlarge the class of problems that can still be solved to some extent, but eventually they too break down on larger problems. The infamous *curse of dimensionality* (Bellman, 1957) denotes the fact that the problem size grows quickly with the number of important dimensions. If a problem state is described by n bits of information, the number of possible problem states grows exponen-

⁴Details of the game can be found in the paper by Romein and Bal (2003) in which the complete game is solved, i.e. an optimal evaluation for all states has been computed. It contains a description of the game and pointers to computer science literature on AWARI. This particular solution got some attention from the Dutch news papers, see DE VOLKSKRANT 17th Aug. 2002 and NRC HANDELSBLAD 14th/15th Sep 2002.

⁵However, in this very special case, researchers for the University of Utrecht (see Romein and Bal, 2003) have computed the complete value function, taking up around 778 gigabytes. The computation was done by a 144-processor, parallel computer system, equipped with 72 GB main memory and a fast interconnecting network. The exact solution technique is described as *retrograde analysis*, and is actually an instance of *decision-theoretic regression* techniques described in this chapter, where a backward state space search technique from the goal states is used to find values of all other states. This highly optimized system computed the value function in about 51 hours, using 10^{15} bits of communication and terabytes of disk I/O. Of these 51 hours, about 15 hours were spent on positions in which all 48 stones are present in the position. The final result gave significant insight in the game, and especially in the strategies of AWARI-playing computer programs. For example, the winner of the Computer Olympiad 2000 made 13 wrong (non-optimal) moves in the finals.

tially in n , in this case 2^n . In general, a problem state is defined as the *Cartesian product* of all dimensions, such that the total number of states exceeds 2^n . Even though some algorithms might have computational complexities *polynomial* in the number of states, the fact that state space sizes grow *exponentially* in the number of features, makes these algorithms infeasible for most practical problems. In fact, even algorithms that have a computational complexity which is *linear* in the number of states will not scale up due to the size of the state space. The storage, learning and needle-in-a-haystack problems all have in common that the algorithms presented so far work *at the level of individual states and actions*, and their numbers quickly grow too large.

3.1.3 Representational Aspects

The level of individual states and actions is called the *flat* MDP. In this flat model, all components are represented in tables, and learning algorithms lookup and backup values of *individual* states and actions. In other words, the *representation* of the problem is as large as the problem itself. Chapter 2 is – in that respect – purely concerned with the *algorithmic aspects* of computing solutions for MDPs. All concepts and algorithms in that chapter are about efficiently computing an optimal value and action *for each state separately*. Starting from this chapter, we will be more concerned about the *representational aspects* of RL.

A solution to the fundamental problems with huge state spaces is to make use of the *inherent structure* in the problem, i.e. the MDP in this case. Arguably, exploiting the structure of the world to solve problems is what computer science is all about. Its literature is filled with smart tricks and techniques for exploiting certain kinds of structure that are found in various problems. *Structured representations* can compactly represent states, actions and policies, often polynomial-sized with respect to the number of variables and actions describing the problem. Many problems show various kinds of structure, regularities and symmetries that can form the basis for *generalization* and *abstraction* purposes. Once an optimal action a for state s is learned, this action a is often optimal for states "similar" to s too. The rotation symmetries in TIC-TAC-TOE (see Figure 3.1, p. 70) effectively decrease the number of states to 25%, such that the number of values to be learned decreases by the same amount.

The main reason for studying abstraction in AI and computer science in general, and in the context of MDPs in particular, is to decrease the amount of computation. Abstractions enable more *compact* solutions, less computation and because of this the scaling up to larger problems. DP and RL algorithms need many iterations of computation, and when more compact models are used, less parameters (e.g. state values) have to be learned. Rich representations have the advantage that they can handle more complex, more accurate and more comprehensible KR. However, more complex representations require increased computational efforts, which is a major concern from a computer scientist's point of view. For example, theorem proving in first-order logic is provably more complex in terms of computational complexity than in propositional logic. Finding and using the *right abstraction level* is a trade-off between computational costs associated with the particular KR formalism, the possibility of learning the right task, and the added benefits of the abstraction levels such as *convergence speed* and *sample complexity* (Kakade, 2003).

Besides the efficiency gains and with that the possibility of scaling up to larger problems, there is an additional reason for abstraction in MDPs. From a KR *point of view*, modeling systems at the most fine-grained level of detail is cumbersome and not intuitive.

For example, when specifying a transition matrix for an agent wandering around in a grid, it is cumbersome to specify each possible transition between grid positions separately. A better way is to give each grid position an X and Y coordinate and specify rules such as: **if** $xpos$ is n and action is east **then** $xpos := xpos + 1$ (in which $xpos$ is the agent's current X -position in the grid). Typical planning formalisms such as STRIPS (Fikes and Nilsson, 1971) enable compact specifications of problems. Various other modeling languages exist which can be also used in specifying parts of MDPs in compact form. These languages enable the incorporation of domain knowledge and – after learning – the *extraction* of acquired knowledge in comprehensible form and possibly *transfer* to related tasks. Furthermore, they can be used for *structured versions* of DP algorithms. In this chapter we will see various forms of KR issues in modeling (and solving) MDPs and in subsequent chapters we will encounter the use of powerful representational languages such as relational and first-order logic.

What is considered a *good* representation is dependent on the task at hand, and the context in which it is used. We will see that abstraction levels in the MDP context are always related to the rewards and transition probabilities, such that a *good* abstraction will maintain those properties that enable the agent to solve the original, flat MDP. Good representations for MDPs will generally **i)** be more *compact* than the flat MDP, i.e. contain less components and tunable parameters, **ii)** decrease learning time of algorithms for computing value functions and policies, **iii)** make modeling of the MDP easier, and **iv)** possibly make *transfer* of the final solution to related problems possible.

3.2. The Essence of Abstraction

Abstraction is a fundamental activity in human perception, conceptualization and reasoning. Strongly put, without abstraction we would be overwhelmed by an information avalanche, obstructing every opportunity to make sense of the world around us. The intuitive meaning of abstraction is to *neglect* information *not necessary* for the activity at hand, while *focusing* on the aspects that *do matter*. The core idea of abstraction is to reformulate a problem into reduced form, to decrease complexity in some distinct way. Abstraction is thus a fundamental mechanism for saving cognitive efforts, by offering a "higher" level view of our physical and intellectual environment.

Let us consider an intuitive example (see Figure 3.2). When travelling to a conference in a city you have never been to, obtaining a good map of the city metro usually suffices to get you to your hotel and your conference location. The metro lines laid out over the city map, highlighting all the stations, abstract away all irrelevant details necessary to plan your journey through town. What color the walls are, what the metro actually looks like, where stairs might be and where you can buy a sandwich, are all details not needed for deciding how to travel. However, when you are in the metro station, details concerning stairs, platforms, signs, the metro, entrances and all that, are important to get you in the right train. Depending on your *current task* (planning your travel through town or getting in the right train) various details are either relevant or can be neglected. This is the core of using abstraction in everyday life.

In computer science, abstraction is a key element in tackling complex problems. It is strongly related to KR, i.e. every KR scheme essentially *is* an abstraction of the problem to be solved or the real world. However, abstractions are usually linked to the process of mov-



Figure 3.2: a) A metro plan of Paris. b) A sign of a metro station in Paris.

ing between *different levels of description* of the same problem. In fact, *successes often hinge on looking at a given problem from different points of view*⁶. Good examples of abstraction in computer science are programming languages. The use of *high-level* languages such as JAVA or C++ is essential to implement complex software systems, although – in principle – one could program the same systems in assembler language. The right *level* for implementing a web-browser is to think in terms of colors, fonts, lines, margins, internet addresses and pictures. Inappropriate levels would be to think of register values of a graphics card to implement a function for displaying a picture in the browser window, or the low-level bit transfer of a network card when connecting to a web server. Programmers use modules, libraries and functions to abstract away details not important at that level during implementation and to enable *reuse* of useful subcomponents of the system. *Object-oriented* programming languages abstract away details into *objects* that can be moved around and used in the software system. A next step is the use of *agents* (Wooldridge and Jennings, 1995), which can be seen as objects with increased autonomy⁷.

3.2.1 Knowledge Representation

As mentioned, *knowledge representation* (KR) (Markman, 1999; Sowa, 1999; Brachman and Levesque, 2004) is intimately related to abstraction. A definition of *representation* is

⁶Finton (2002) used a suitable quote in his PhD thesis about this:

”At PARC we had a slogan: ‘Point of view is worth 80 IQ points.’ It was based on a few things from the past like how smart you had to be in Roman times to multiply two numbers together; only geniuses did it. We haven’t gotten any smarter; we’ve just changed our representation system. We think better generally by inventing better representations; that’s something that we as computer scientists recognize as one of the main things that we try to do.”

Alan Kay

⁷Since the heightened attention for *agent-based* systems, a debate is ongoing about *what* exactly are the *characteristics* of an agent, and furthermore, what *distinguishes* them from related concepts such as *objects*, *programs* and *processes*. Useful characteristics are *autonomous*, *pro-active*, *adaptive*, *mobile*, *embodied*, *conversational*, *rational* and many more. For this chapter, it suffices to see agents as a next step in a long line of abstractions used in software engineering. But see (Ferber, 1999; Weiss, 1999; Wooldridge, 2002) for pointers to the literature.

philosophically as vexing as that of *knowledge* (Brachman and Levesque, 2004, p.3). Generally speaking, representation is a relationship between two domains, where the first is meant to "stand for" or "take the place of" the second. Usually, the first is more concrete, *abstract* or accessible than the second. In this book we do not deal explicitly with the difficult topic of *perception*, i.e. how representations arise from raw, real-world data. For example, in using an intelligent system for identifying people's faces in a crowd, a large part of the problem is obtaining data from camera images, processing this data and transforming the data into features that can be fed into a recognition system. For the topics described in this book, we assume that the problem is already defined by a set of *features* describing the problem in some particular representation language. The abstractions and algorithms we describe all start from an initial problem formulation. As a consequence, we do not describe, nor deal with, the difficult problem of *symbol grounding* (Harnad, 1990). The symbol grounding problem essentially deals with the question: *where do the representations come from?*, i.e. how a representation relates to its *real* counterpart. For example, grounding the concept *chair* in the perceptual input of a robot's sensors comes down to the question of what a chair actually is. Does it need to have arms, or three or four legs, and does the exact shape matter? These are all difficult questions, from various viewpoints such as cognitive science, philosophy and computer science. Margolis (1999) and Gärdenfors (2000) include recent overviews of different views on this topic. Brooks (1991) in the early nineties, challenged the tenet of GOFAI⁸ stating that finding a good abstraction of a problem was the *essence* of intelligence and was, in many AI systems, done by the researcher himself. In fact, Brooks talks about *abstraction as a dangerous weapon*⁹. Recent directions that are in line with this view of incorporating perception and motor skills as an integral part of the whole AI approach are *embodied intelligence*¹⁰ (Pfeifer and Scheier, 1999; Clark, 1997; Ziemke, 2000) and *behavior-based* approaches (Arkin, 1998). See for an opposite view on representations (Markman and Dietrich, 2000a,b). We agree that in generally intelligent systems, this should all be learned by the system, and not designed by hand, roughly in spirit of the work on *autonomous concept formation* (e.g. see Drescher, 1991; Thornton, 2000; de Jong, 2000). In this book we confine ourselves to the task of learning and using representations using a priori defined representation languages and paradigms. Among these are languages such as propositional and first-order logic, and paradigms such as the connectionist, evolutionary and symbolic ones. In Section 3.3.3 we will describe propositional KR schemes in somewhat more detail, and in Section 4.2 we will describe first-order formalisms.

⁸Good Old-Fashioned AI, a term for a classical view on AI that is focused on symbolic computation.

⁹From (Brooks, 1991, p.2): "Typically, AI 'succeeds' by defining the parts of the problem that are unsolved as not AI. The principal mechanism for this partitioning is abstraction. Its application is usually considered part of good science, not, as it is in fact used in AI, as a mechanism for self-delusion. In AI, abstraction is usually used to factor out all aspects of perception and motor skills".

¹⁰But see also the new direction of *predictive representations of states* (Littman *et al.*, 2001) in which the concept of *state* is abandoned and a more realistic *stream of observations* is considered.

3.2.2 Definitions and Theories of Abstraction

In the AI community abstraction¹¹ can be defined somewhat more formally as a *mapping between formalisms (i.e. representations) that reduces the computational complexity of the task at hand*. It is often associated with a transformation of the problem representation that allows to solve the task more easily, or, *with reduced computational effort*. For example, changing the representation of a problem can make theorem proving or reasoning more efficient, i.e. the desirable answer is produced with less computational effort, using a smaller number of inference steps. Precise definitions of "abstraction", distinguishing it from the all-encompassing notion of reformulation, exist in certain specific contexts, but a precise, universal definition of "abstraction" is not yet available (see Giunchiglia and Walsh, 1992; Holte and Choueiry, 2003; Zucker, 2003, for overviews). Abstractions can be distinguished along several dimensions (Zucker, 2003). Examples include *domain hiding*, where parts of a problem are abstracted away and *domain aggregation*, where parts of the domain are blended together into complex concepts (e.g. a *keyboard*, *PC tower* and *monitor* make up the concept *computer*). Two general definitions of a common view upon abstraction in AI are the following (Zucker, 2003):

DEFINITION 3.2.1 ► An **abstraction** is a change of representation, in the same formalism, that hides details and preserves desirable properties. A **reformulation** is a change of representation, from one to another formalism, preserving the quantity of information involved.

Although this definition captures the core concept of abstraction, the fact that it requires the *same* formalism makes it somewhat restrictive. In many cases, the formalism (i.e. the representation) is changed as well. A *reformulation* into a simpler, or even more expressive formalism to perform abstraction covers a somewhat larger class of transformations. In the remainder of this book, we will loosely use the word *abstraction* for both. In the definition, three important concepts play an important role although their exact meaning is dependent on the specific context in which they are used. These are *simplicity*, *detail* and *desirable properties*. They are difficult to define in general terms, but in the next sections, these aspects can get a more concrete implementation in the context of MDPs.

3.2.3 Representation Change

In the previous we have encountered some of the properties of abstractions. They are useful in identifying *what* abstractions are and *why* they are useful. A next step is to view upon the *process of changing* representations in problem solving (Korf, 1980), reasoning and machine learning (Saitta, 1996). The general idea of changing the statement, or representation, of a given problem is called, in AI, *change of representation* or *reformulation*. Holte and Choueiry (2003) give examples (outside machine learning) in planning and problem solving, in constraint processing and in reasoning about physical systems. The intuitive idea is that a representation change is an abstraction if the computational cost to solve a class of problems is significantly reduced.

¹¹Although basically all AI research deals with many kinds of abstraction – and thus all main conferences and journals in this field deal, implicitly or explicitly, with it, there is a special forum devoted to abstraction. The bi-annual *Symposium on Abstraction, Reformulation and Approximation (SARA)* shows a wide range of abstraction research.

The classical paper by Korf (1980) defines two orthogonal dimensions along which representation changes occur, the information *structure* and the information *quantity*. Using these dimensions Korf (1980) distinguishes between *isomorphisms* and *homomorphisms*. An isomorphic transformation changes the information structure of a representation while leaving the information quantity fixed. For example, one can represent the game of TIC-TAC-TOE in the usual form, but also in the form of a card game. In this form, nine cards (numbered 1 to 9) are put on the table face up between the players. Each player alternately selects a card, and the winner is the first one obtaining three cards that sum up to 15. Thus it is the structure of the problem (the representation) that changes, but not the essence of the game¹². Isomorphic transformations contain the same information as the original problem, but the new representation might make a big difference in how it can be used for learning and reasoning. A homomorphic transformation on the other hand, alters the information quantity but leaves the information structure unchanged. This is the key component for a recursive solution to the *Towers of Hanoi* problem (see Korf, 1980). Homomorphisms often induce *equivalence* classes of elements in the original representation that cannot be or do not have to be distinguished in a given problem.

Reformulations of problems (either isomorphic or homomorphic) require three distinct components: **i)** a language for expressing representations, **ii)** a language to describe transformations of representation and **iii)** and interpreter to apply a transformation to a representation and compute the resulting representation. Points in the space of representations can be described by expressions in a (symbolic) representation language. The operators in this space can travel along the information quantity or information structure dimensions. Each point in this space is a representation¹³. Reformulation is a natural phenomenon in machine learning systems. Broadly speaking, any concept learning system can be thought as undergoing knowledge representation changes during its activity, e.g. in inducing general hypotheses from examples expressed in a domain specification language. However, one has to distinguish between abstraction and *generalization*, which have complementary properties and goals though. Generalization is the main mechanism for hypothesis *formation*, whereas abstraction provides a mechanism for *representing* (Saitta, 1996). Related concepts that are of interest are *approximations* and *aggregations*. Approximations are abstractions in which the information quantity is lowered to the extent that optimal performance is sacrificed for a gain in computational efficiency. Resources such as available computation time and memory can be limited (see Russell, 1997) such that one has to *satisfice*¹⁴. Some problems are even so large that approximations are the only way to compute a solution at all. An aggregation is an abstraction in which a *set* of several entities is represented by a single entity. Aggregations are commonly used in the MDP framework, for example in *state space aggregations*.

Automated representation change is highly desired for truly intelligent agents. If the agent can develop its own view, i.e. representation, of the environment, it is less dependent on a priori knowledge of a designer. Additionally, the agent's view can be more *specialized*

¹²For human players, the original definition is more familiar and easier to play (verified by the author).

¹³(Korf, 1980, 74): "Changes of representation are not isolated 'eureka' phenomena, but rather can be decomposed into sequences of relatively minor representation shifts."

¹⁴Approximation is one form of abstraction in which information is lost. One of Richard Sutton's AI research principles is "Approximate the solution, not the problem". What he means by that is that approximation should always be relative to *resource bounds* (e.g. available computation time, memory, reasoning capabilities etc). In the case of infinite resource bounds, the solution should be optimal.

to the task it is solving such that it is more *grounded* in the environment it is currently in. Adaptive representations are also a solution to non-deterministic and changing environmental dynamics. Automated representation change is a hard problem, and should be aided by as much *bias* and *knowledge* about the environment as possible. Most of the *constructivist* approaches build new representations based on already acquired ones such as in the *many-layered learning* approach by Utgoff and Stracuzzi (2002). New *concepts* (e.g. propositional sentences over state features) are built on top of concepts acquired in earlier learning periods. Similar constructs under the name *predicate invention* have been proposed in the *inductive logic programming* (Muggleton and De Raedt, 1994) community. Learning truly *new* concepts and representations in artificial intelligence system seems an ill-defined problem¹⁵ as all concepts will eventually be grounded in the perceptual input.

Different abstractions of the same problem form so-called *abstraction levels*. The original problem description can be considered the most complete and accurate level, and all other abstraction levels can be reached along the information quantity and information structure dimensions, along the lines of Korf (1980). The original problem definition can be seen as a kind of *semantic layer* of all abstraction levels defined for this problem. Each abstraction level is a particular *representation* of that problem. *Representation theorems* can ensure that the newly introduced abstraction layer models (exactly or approximately) the underlying problem.

Changing the representation of a problem can have a huge impact on the computational efforts needed to solve this problem. However, representation change comes with a price. Interestingly, Zucker (2003) hints at the fact that a *no-free-lunch theorem* might exist for abstraction too. The *no-free-lunch* theorem in machine learning (Wolpert, 1995) says, roughly, that there are no learning algorithms that are always optimal. For every learning algorithm, there exist problems for which it performs worse than others¹⁶. A similar theorem for abstractions would say that there is no universally optimal abstraction method. Wrongly chosen abstractions might actually makes performance worse. For example, there are *dynamic Bayesian network* (see Section 3.5) representations of MDPs that are exponential in the number of features. There are several aspects that are related to the *utility* of representation changes (Zucker, 2003). This stresses the importance of having the right representation for the right task.

3.3. Abstraction in the MDP Setting

General definitions and theories of abstraction deal with the questions what abstractions are, why they are useful, and how they can change. In concrete situations, aspects such as *simplicity* and *desirable properties* (see Section 3.2.2) can be given a concrete meaning. In this chapter we describe the use of abstraction in a more concrete context, in the MDP framework. Many realistic domains modeled as MDPs exhibit considerable structure that can be modeled explicitly and exploited so that typical problems can be solved efficiently. The aim here is twofold. First, abstraction enables more *compact representations*, thereby providing more efficient solutions to the problems with huge state spaces mentioned in

¹⁵This presupposes a definition of *new* which will difficult to define without resorting to concepts such as *creativity* and it involves defining *heureka* moments in an otherwise gradual development.

¹⁶Saying that a classification algorithm is good actually comes down to saying that the inductive bias matches the data well.

Section 3.1.2. Second, abstraction provides means for the incorporation of *structure* in both *modeling* and *solution* techniques used in the context of sequential decision making. Abstractions in MDP add a *structural* dimension to the algorithms described in the previous chapter. Many *scalability issues* have been extensively addressed in the MDP literature (Littman and Majercik, 1997; Littman *et al.*, 1995; Bertsekas and Tsitsiklis, 1996; Geffner, 1998; Boutilier *et al.*, 1999; Goldsmith and Sloan, 2000; Feng, 2004), and here we will highlight important directions.

Knowledge representation schemes play an important role in exploiting structure in MDPs. An important distinction here is that between structured *representations* and structured *operations*. The first is about compact representations of structures such as state spaces and value functions. For example, a neural network can compactly *represent* a value function in terms of an architectural bias combined with a set of parameters. Structured *operations* refer to the fact that, in the presence of structured representations of e.g. a transition function, operations such as Bellman backups can be defined in *structural form* too, i.e. they can be defined on an abstraction level such that they update values for abstract states (i.e. sets of states). Several algorithms make use of this (see Section 3.5) and in the remainder of this book we will see examples in first-order logic. In this chapter we will refer to *structure* as all *intensional representations* used for policies, value functions, states, actions, etc. *Parameters* on the other hand, are all other *tunable components* of the agent that stand for values, concrete actions, probabilities or architectural parameters (e.g. the weights in a neural network).

1	2	3	4	g_1
5	6	7	8	g_2
		9		
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24
25	26	27	28	29

Figure 3.3: An example *grid world*.

Abstractions in the MDP framework introduce a trade-off between *sample complexity* and *computational complexity* of algorithms (see also Kakade, 2003; Littman *et al.*, 2005). The algorithms in the previous chapter are mainly concerned with the sample complexity, with some exceptions such as prioritized sweeping which involves complex backups. The sample complexity of RL algorithms denotes the *amount of samples* needed to obtain near-optimal performance. The aim of abstraction is to decrease sample complexity by offering more compact abstraction levels, creating simpler problems and less parameters that have to be estimated. Computational complexity expresses the *amount of work needed per experience*. Generating and adapting structures, dealing with structural credit assignment and the use of structured operations, all increase computational complexity of algorithms in this respect.

Model-free algorithms are more concerned about sample complexity, whereas for model-based algorithm – that do not experiment with the MDP – the computational complexity is the bigger issue.

Several other dimensions are important in addition to these issues. In the introduction to the previous chapter it was explained that the MDP formalism allows for variation in the *quality* of the solution. Abstraction levels may give rise to various forms of *satisficing*, i.e. learning *non-optimal* policies in the face of resource bounds (e.g. see Russell, 1997). Abstractions over an MDP can be pure *isomorphisms*, such that an optimal policy for the original MDP can still be found, however possibly with a reduced computational effort. On the other hand, abstractions can also be *homomorphisms* or *approximations*, such that reasonable policies, though possibly not optimal, can be found quickly and with reduced

effort. In fact, abstraction induces a whole range of solutions, differing in compactness, learning speed and time, final policy quality (e.g. in terms of average reward) and computational resource needs. For example, approximations of value functions (see Section 3.6) do not have to be perfect in order to derive an optimal policy. There is a trade-off between the computational efforts to obtain a certain precision in the value function approximation and the quality of a policy derived from this value function. A different kind of trade-off can be found in *hierarchical* methods (see Section 3.8). Here, a hierarchical decomposition of both policies and value functions can prevent finding an optimal policy for the original problem, though in return, they allow for faster learning, modular structure and possible reuse of partial solutions.

3.3.1 Dimensions of MDP Abstractions

Broadly speaking, there are five categories of abstractions for solving MDPs. One can define abstractions of the MDP itself, in *abstract state spaces* or *structured MDP models*, in *value functions*, in *policies* and in the *temporal structure* in problems (i.e. action sequences and task structure). Although presented here as *orthogonal* directions, often multiple abstractions are combined (see Steinkraus and Kaelbling, 2004; Fitch *et al.*, 2005; Littman *et al.*, 2005).

For example, abstract value functions are often used to induce abstract policies, and many temporal abstractions are combined with state space abstractions (e.g. see Dietterich, 2000a). An important property of abstraction dimensions is whether the abstraction levels are *fixed* or *adaptive* during learning. Fixed abstraction levels create a stable environment for parameter and value learning algorithms. However, adaptive abstractions allow the algorithm to find an abstraction level that is adapted to the current task, and they free the programmer of supplying it. However, this comes with the increased computational complexity of structural adaptations.

Consider the *grid world* depicted in Figure 3.3. This environment has 29 identifiable states (grids 1–29), and 2 additional goal states (g_1 and g_2). Furthermore, there are two *wall* parts (the black squares). The agent is started somewhere in one of the states 10–29 and the goal is to get to one of the goal states. This task is episodic, which is modeled using absorbing goal states. The actions available to the agent in each of the grid positions are north, south, east and west which take the agent one position into that direction with probability 0.9 and with probability 0.1 nothing happens, i.e. the agent stays in the same state. However, moving from the edge of the grid, or moving against the wall, is not allowed, i.e. the agent stays in the same position. For example, after performing action north in state 11 the agent will still be in state 11 and performing action east in state 24 results in the agent staying in state 24. Moving into a goal position will give the agent a reward +1, e.g. performing action east in state 8 will result in a transition to g_2 and a reward +1 (with probability 0.9). All other transitions generate a reward 0. The discount factor γ is 0.9. In Figure 3.4 the structure (*state graph*) of the MDP modeling this grid

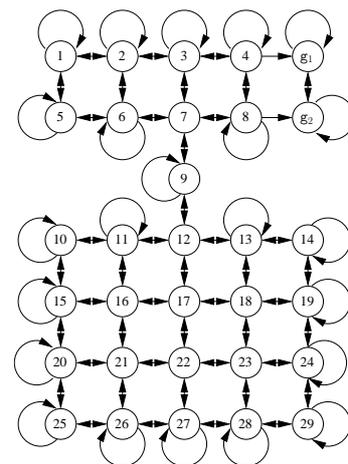


Figure 3.4: A *state transition graph* of the grid world from Figure 3.3.

world is depicted. The MDP is specified as $\langle S, A, T, R \rangle$, where $S = \{1, \dots, 29, g_1, g_2\}$ and $A = \{\text{north, south, east, west}\}$. The transition function T contains all transitions (depicted as bidirectional lines) such as $T(22, \text{east}, 23) = 0.9$ and $T(15, \text{west}, 15) = 0.1$. The reward function R defines $R(4, \text{east}, g_1) = 1$, $R(8, \text{east}, g_2) = 1$ and for all other transitions $R(s, a, s') = 0$ whenever¹⁷ $T(s, a, s') > 0$. In the following sections we will give examples of different kinds of abstraction using this example environment.

3.3.1.1 STATE SPACES AND THE MDP MODEL

A fundamental type of abstraction is to abstract the flat MDP itself. Using the properties of the MDP one can *aggregate* parts of the model, such as groups of states, into larger components. The end result is an *abstract* MDP for which standard algorithms from Chapter 2 can be used to compute optimal (abstract) policies. The abstract MDP is typically a *homomorphism* of the flat MDP that ensures the possibility of learning a policy that is optimal with respect to this flat MDP too. Among many abstraction-based algorithms there are several approaches that deal explicitly with the intrinsic properties of model-minimization in *model-based* approaches (Givan *et al.*, 2003), *model-free* approaches (Finton, 2002) and *hierarchical approaches* (Ravindran, 2004).

Model-Minimization. Aggregation can be based on the transition and reward functions, or based on value functions. Model-based methods using *stochastic bi-simulation* measures (Givan *et al.*, 2003) or based on ignoring of irrelevant variables (Dearden and Boutilier, 1997) are described in Section 3.4.1.

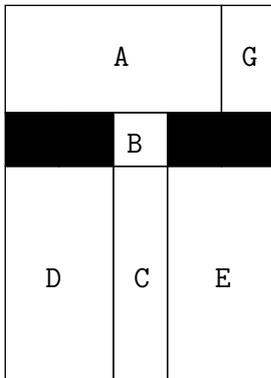


Figure 3.5: *State abstraction in the grid world from Figure 3.3.*

In Section 3.8 we will encounter a model-minimization approach by Ravindran and Barto (2001) based on homomorphisms that was extended to minimizations using hierarchical abstraction (Ravindran, 2004). As an example of what can happen using state aggregation to get a more compact model, see Figure 3.5. In this figure, five *regions* in the state space are depicted. The regions A–E are *aggregations* (or, *abstract states*) that, together with an *abstract goal region* G, make up a new abstract state space that is depicted in Figure 3.6. For example, the states 1–8 from the original, flat MDP (see Figure 3.3) are *aggregated* into the *abstract state* A.

This makes sense, because for an optimal policy, the agent does not have to distinguish between the states 1–8 because in all these states, the optimal action is east. Furthermore $\pi^*(B) = \pi^*(C) = \text{north}$, $\pi^*(D) = \text{east}$ and $\pi^*(E) = \text{west}$. Note that the transition probabilities between the states in aggregated model are now dependent on the exact state (in the flat model) the agent is in. If the agent performs east in state 11, with probability 0.9 a transition is experienced from state D to state C. However, after performing the same action in state 10 (which is also in D), with probability 1 the agent stays in state D, although with probability 0.9 the agent will move *within* state D to state 11. In essence, the new, abstract state space is not Markov anymore: it depends on the history of visited (concrete) states whether the transition of D to C is made or not.

¹⁷If T and R are represented by full matrices, R is defined for the complete Cartesian product $S \times A \times S$, but for convenience we only note rewards for transitions that actually exist in this MDP.

Adaptive Resolution. Model-free decompositions of MDPs are usually called *adaptive resolution methods* and will be described in Section 3.4.2. In the absence of a model of the MDP, useful abstractions can be found based on statistics gathered in the environment. Statistical tests can be used to decide how to aggregate components. Several *state-splitting* criteria have been defined for online, incremental decomposition, for example based on *trees* (Chapman and Kaelbling, 1991) and *cognitive economy* (Finton, 2002). These splitting criteria can be used to induce similar abstract MDPs such as depicted in Figure 3.5. One starts with a model that only distinguishes between goal states and other states, and during learning abstract states are split, which generates increasingly more fine-grained state abstractions.

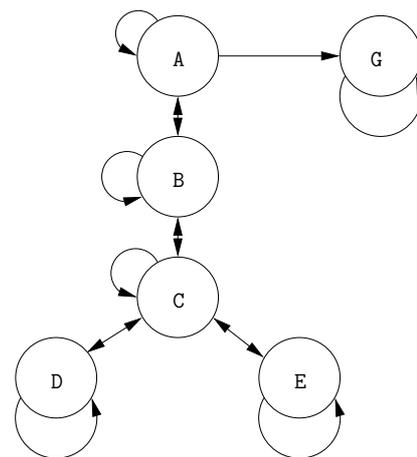


Figure 3.6: A state graph for the abstract MDP from Figure 3.5. The arrows depict non-zero transition probabilities.

Factored Representations. Another line of abstraction of MDPs is by using *factored representations* (Boutilier *et al.*, 2001) of state and action spaces, as well as transition and reward functions, see Section 3.5. Compact representations such as *dynamic Bayesian networks* (DBN) (Dean and Kanawaza, 1989; Ghahramani, 1998), *trees* and *algebraic decision diagrams* can be useful in two distinct ways. The first one is to obtain a compact *representation* of the MDP and its dynamics; factored representations are usually exponentially smaller than the original MDP. The second benefit is that *structured versions* of policy and value iteration can be defined, such that algorithms and value backups work *on the level of abstract states*. Factored representations can also be used in model-free contexts (Sallans, 2002; Sallans and Hinton, 2004). In our grid world example, one can replace the *discrete symbols* 1–29 by a *coordinate system* using X and Y coordinates as in Figure 3.7. The lower-left corner is position (1, 1) and each grid position can be represented using $(xpos, ypos)$. We know that positions (5, 6) and (5, 7) are goal states, and that the initial state is sampled as $(xpos, ypos)$ where $xpos \in 1 \dots 5$ and $ypos \in 1 \dots 4$. Now transitions between a state $(xpos, ypos)$ to a next state $(xpos', ypos')$ after performing action a can be compactly represented by a set of rules such as **if** $xpos < 5$ **and** $a = \text{east}$ **then** $xpos' := xpos + 1$ with probability 0.9. A representation such as a DBN can represent these rules more compactly in terms of probabilistic dependencies between $(xpos, ypos)$, $(xpos', ypos')$ and the current action.

Parallel Decompositions. A last type of abstraction deals with *parallel decompositions* of MDPs. Sometimes an MDP can be decomposed into several MDPs which can be solved independently, each optimizing a particular, independent contribution to the reward function. In such a parallel decomposition a solution to the overall problem can be seen as the parallel execution of all actions suggested for the component MDPs. Several decomposition techniques exist (e.g. Singh and Cohn, 1998), and they are particularly useful in multi-agent systems where such a decomposition naturally translates to a decomposition into agents (e.g. see Kok, 2006). Parallel decompositions are beyond the scope of this book, but in Chapter 7 we will return to multi-agent systems, in the relational setting.

The goal of decomposing an MDP is to find a compact abstraction level such that value learning can be performed on this level, so that value functions and policies found on this

level perform well on the original, flat MDP. Virtually all methods that adapt structures online decompose in a *top-down* fashion, i.e. one starts with one model component and iteratively splits model components into multiple components until a suitable abstraction level is found. After each iteration, the abstraction is *refined*. In model-free algorithms a suitable abstraction level is usually found *during* learning, whereas for model-based methods this can also be performed *before* learning.

3.3.1.2 VALUE FUNCTIONS

(1,7)	(2,7)	(3,7)	(4,7)	(5,7)
(1,6)	(2,6)	(3,6)	(4,6)	(5,6)
		(3,5)		
(1,4)	(2,4)	(3,4)	(4,4)	(5,4)
(1,3)	(2,3)	(3,3)	(4,3)	(5,3)
(1,2)	(2,2)	(3,2)	(4,2)	(5,2)
(1,1)	(2,1)	(3,1)	(4,1)	(5,1)

Figure 3.7: A factored state representation of the grid world example from Figure 3.3.

State value functions V and state-action value functions Q are *real-valued functions* and can be represented more compactly using *parameterized* structures. The number of parameters is often exponentially smaller than the number of states or state-action pairs. Popular choices are *linear combinations of basis functions* and *neural networks* (Bertsekas and Tsitsiklis, 1996). An advantage – especially for model-free learning – is that for most architectures parameters can be estimated in an incremental, online fashion. Function components can be fixed, i.e. the abstraction level is fixed, but they can also be adapted (and their number increased) during learning. Adapting the basis functions during learning is a hard problem. Often, computation switches between value learning periods and basis function adaptations. In our grid world example, a *featural representation* in terms of x_{pos} and y_{pos} can form the basis for function approximation. Here, the value of a state can be represented (and learned) as the linear combination $V = \theta_1 x_{pos} + \theta_2 y_{pos}$ where θ_1 and θ_2 are tunable parameters, such that the state value table is represented by just these two parameters. In Section 3.6 various methods for value function approximation will be discussed, both with fixed and adaptive basis functions.

3.3.1.3 POLICIES

An abstract policy is a structure mapping states to actions. Various structures can be considered and algorithms exist that search directly in the space of policy abstractions. For example, a neural network can represent a (stochastic) policy and policy search then comes down to searching for the right network. There are three general directions in policy search. The first is the use of *evolutionary algorithms* (Moriarty *et al.*, 1999) which mimic biological evolution to find good policy structures. The second direction consists of *gradient-based* approaches. In this, an initial policy abstraction is given at the beginning of learning. During learning, the parameters of the structure are adapted based on the reward in-take during learning. In both directions, the focus is on learning policy abstraction, usually without explicitly representing value functions. However, policy gradient methods typically adapt fixed structures whereas evolutionary methods learn the abstraction itself. The third direction aims at finding suitable examples to train policy structures using *classification* algorithms (e.g. Lagoudakis and Parr, 2003). In our grid world example, one can search in the space of policies $\{1 \dots 29\} \rightarrow \{\text{north, south, east, west}\}$ or, using a featural representation, in the space of policies $\{1 \dots 5\} \times \{1 \dots 7\} \rightarrow \{\text{north, south, east, west}\}$.

Such policies can be more compact than value functions because they only have to store optimal actions (which can be the same for many states) instead of values for all states. Obviously, the *representation* of functions implementing these policies can take various forms. All three policy search directions are described in Section 3.7.

3.3.1.4 TASK STRUCTURES AND HIERARCHIES

Another direction in MDP abstraction consists of *hierarchical* methods. These methods focus attention on the *sequential* and *temporal* aspects of a task. A sequential decision problem can be decomposed into various subproblems that can be solved separately. Sequences of actions can be abstracted into *temporally extended actions*, which can be considered as *sub-skills* in the overall task. Many hierarchical approaches define a set of temporally extended actions and value learning can make use of these abstract actions. Another type of methods decomposes the state space into separate *regions* and learns sub-skills to travel between these regions. Overall, these methods abstract from sequences of actions, but combinations with other abstraction types (e.g. state abstractions) are commonly used. Take as an example state 9 in the original grid world (see Figure 3.3 on page 82) example (which is abstract state B in the abstract MDP at page 84) of this section (see Figure 3.6). This state functions as a kind of *bottleneck* state between the upper and lower part of the grid. All optimal policies for states 10–29 lead through state 9. Having identified this, one can define 9 as a *subgoal*. An optimal policy for the lower part of the grid will first move to 9 and then progress to the goal. This policy is easier to learn, because it contains fewer actions and is defined for a smaller *sub-MDP*. Hierarchical methods identify these subgoals and learn *partial* policies (or, sub-skills) for getting to these subgoals. One can identify more subgoals in this grid. For example, all optimal policies from a state in abstract state D will lead to a state in abstract state C) before moving to state 9. Thus, states in the abstract state C can function as yet another subgoal for the regions D and E. Note however, that it depends on how we setup the subgoals on whether the complete policy, i.e. the one that moves from subgoal to subgoal, is optimal in a global sense. For example, moving from state 10 (region D) to state 22 (region C) is optimal on the level of subgoals, but not optimal in general. A faster way from state 10 to region C is to move directly to state 12.

Temporally extended actions, state space decompositions and task hierarchies can be defined before learning, i.e. as fixed abstraction levels. This can be seen as putting a *bias* on the policy space, given by the designer. However, an increasing number of methods aims at *learning* the hierarchical decompositions from samples. Some approaches learn to identify useful *subgoals* or weakly connected components in the state space for the induction of abstractions, while others use a model of the flat MDP. Hierarchical methods are described in Section 3.8.

3.3.2 The PIAGET-Principle

In the introduction to this chapter we have made a distinction between the *algorithmic* and the *representational* aspects of learning sequential decision making. The algorithmic aspect is the topic of Chapter 2 and the underlying principle is *generalized policy iteration* (GPI) (see Figure 2.3 on page 45). The generality of this principle does not exclude the use of abstraction, i.e. the representational aspect. In fact, Sutton and Barto (1998) talk about using value function approximations in GPI. However, to get a full understanding

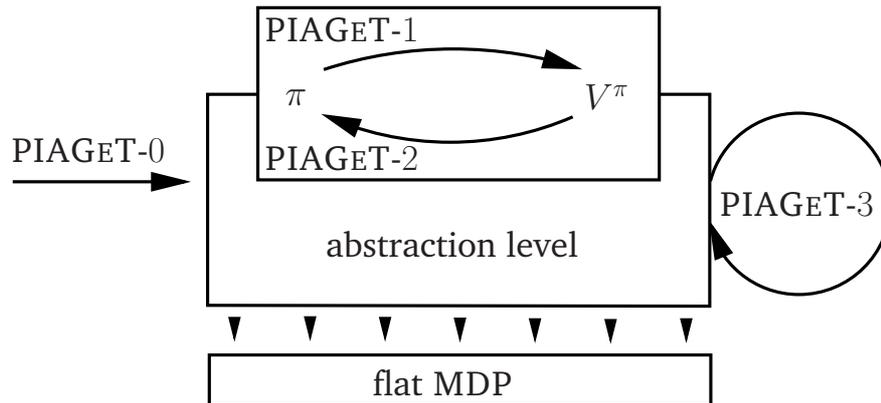


Figure 3.8: Policy Iteration using Abstraction and Generalization Techniques (PIAGET).

of the use of abstraction mechanisms in GPI we will outline a new principle that is based on GPI. This is done by extending GPI to the PIAGET-principle¹⁸, which stands for *Policy Iteration using Abstraction and Generalization Techniques*, and is depicted in Figure 3.8.

There are two ways of viewing the PIAGET-principle. The first is to see it as an *extension* of GPI to the context in which abstractions are used. The general structure of GPI is extended with an abstraction component that functions as a *knowledge representation level* over which the standard GPI can be performed. A second view upon the PIAGET-principle is to see it as an *implementation* of GPI. The general structure is not changed, though it is refined by making the use of abstraction more explicit. In either way, it is our opinion that it gives more insight into some aspects that play a role in using abstraction for learning sequential decisions.

The base level of the diagram consists of the original, flat MDP. The *abstraction level* defines a representation of this MDP that abstracts parts of the MDP, value functions and/or policies. This representation can be viewed as a particular homomorphism (along the lines of Korf (1980)'s work) of the flat model. This abstraction level can be given a priori as a particular bias, based on prior knowledge or assumptions about the domain. For example, a task hierarchy, or a state space decomposition, can be given, or learned, prior to applying learning algorithms. The original GPI process can be seen as working *at the level of abstraction*. For example, when states are aggregated into abstract states, values can be learned for these abstract states and an abstract policy can map these abstract states into actions. The majority of methods keeps a fixed abstraction level during learning. For example, function approximators usually have a fixed, a priori defined, set of basis functions. However, several methods adapt the abstraction level during learning.

Basically, there are two interacting processes in the PIAGET-principle. The base layer is the *structural* part in which abstraction levels are defined and learned. This part deals with the knowledge representation aspects of the MDP. The top layer is the GPI-principle that is responsible for learning values, i.e. parameters of the structural part. The PIAGET-principle clearly separates *structure learning* and *parameter learning*. The parameter learning part is responsible for learning values, actions, transition probabilities in the abstraction layer. However, as was hinted in the previous chapter, when using abstractions it becomes necessary to deal not only with *temporal credit assignment* but also with *structural*

¹⁸Jean Piaget (1896–1980) was a Swiss developmental psychologist. He is well-known for his work studying children, and his (constructivist) theory of cognitive development (Piaget, 1950).

credit assignment. For example, parameterized functions that store value functions might need to adapt *internal parameters* with respect to value updates. The structure learning part of PIAGET is responsible for adapting the abstraction layer. This part uses the information that is obtained in the parameter learning part. This part might have obtained information about whether a state space decomposition has to be refined, or whether useful subgoals can be identified based on reward statistics. The division in structure and parameter learning has much in common with *structural expectation-maximization* (see Friedman, 1998) approaches in statistical machine learning, and algorithms in the field of *probabilistic inductive logic programming* (De Raedt and Kersting, 2003, 2004) where probabilities (i.e. *parameters*) are learned in combination with logical abstractions (see Chapter 4).

The PIAGET-principle separates structure and parameter learning. The majority of algorithms obeys this structure strictly in that either parameter learning is performed on a fixed abstraction level, or that parameter learning is *interleaved* with structure learning. Some algorithms however, such as structured versions of value iteration and policy iteration (Boutillier *et al.*, 2000a), perform both steps in parallel. These algorithms define *structural versions* of value learning algorithms, where each value learning step performs structure adaption.

We can distinguish four classes of algorithms that compute solutions for MDPs in the face of abstraction.

- **PIAGET-0:** The first level consists of methods that construct a compact representation of the MDP, value functions and/or policies, *before* learning. Examples are *feature selection*, *state aggregation* and *model-minimization* methods (see Section 3.4). The abstraction is generated prior to learning an optimal policy *at this level of abstraction* by means of GPI (see PIAGET-1).
- **PIAGET-1:** This level consists of methods that use GPI on an abstraction level either defined a priori or obtained using PIAGET-0. All methods in this level can be seen as applying the algorithms from Chapter 2 over an abstract level, such that only the *temporal credit assignment* problem is present. Learning on this level consists of learning values and actions for abstract states, for example over *state space aggregations*, see Section 3.4.
- **PIAGET-2:** Similar to PIAGET-1 this level consists of methods that learn values over an abstraction level. However, in contrast to that level, PIAGET-2 consists of abstraction levels with *parameters*, e.g. neural networks for *value function approximation*. An *internal structure* of the abstraction level forces learning on this level to deal with both the temporal and structural credit assignment problem, such that value backups on this level are more complex.
- **PIAGET-3:** This level is the most *constructivist* level of learning, and with that the most powerful and complex. This level is an extension of PIAGET-2 level in which the structures defining the abstraction level are themselves subject to change. Structures can for example be extended by moving to a more fine-grained resolution of the model, or by adding structural elements such as adding neurons to a neural network for value function approximation. Methods at this level do *structural adaption* in parallel with value learning. The *structural credit assignment* involves adaption of

parameters *and* structural parts. Usually fixed-length (e.g. in terms of numbers of episodes) interleaving periods of value learning (e.g. on one of the levels PIAGET-1 or PIAGET-2) and structure adaptations are performed. Structural adaptation can for example be initiated by *statistical measures* on errors or value inaccuracies (e.g. in constructive value function approximation, see Section 3.6), due to finding useful *subgoals* (e.g. in hierarchical approaches, see Section 3.8) or as an integrated part of *structural Bellman backups* (e.g. in factored model representations, see Section 3.5).

In addition, a fifth level might be added that is concerned with extracting knowledge from a solution, in order to *transfer* to other problems, though for most of this book, four levels suffice. But, based on the five types of abstraction, and the four PIAGET-levels, one can form twenty different algorithm classes. We will review many of these in the following sections. The most interesting cases are those that function at PIAGET-3, where structure and parameter learning are combined.

3.3.3 Representations in MDP Abstractions

Abstractions in MDPs are generated by knowledge representation formalisms. The methods in this chapter are situated in the *propositional setting* (i.e. the BOOLE setting from Chapter 1). States and actions are described by *features* and abstractions are defined over these features. Let us first briefly define *propositional representations* and highlight different ways of propositional abstraction and generalization.

3.3.3.1 PROPOSITIONAL AND ATTRIBUTE-VALUE REPRESENTATIONS

The representational languages used in the methods in this chapter belong to the class of *propositional* or more general, *attribute-value* (AV) languages. These are simple representations frequently used in machine learning, planning and logic (see further Markman, 1999; van Laer, 2002; Sowa, 1999; Brachman and Levesque, 2004).

In a propositional language, information is represented by *vectors of features*. A feature vector f of length n is a vector $\langle f_1, f_2, \dots, f_n \rangle$ such that each *feature* f_i ($i = 1 \dots n$) can have as value either true or false. For example, the state of a robot with two light sensors that can either detect light or not, can be specified using a feature vector $\langle \text{right}, \text{left} \rangle$ and a concrete situation of the robot sensor states is the state $\langle \text{true}, \text{false} \rangle$ meaning that only the right sensor detects light. In propositional representations, the domain of each features is $\{\text{true}, \text{false}\}$ such that a vector of n features gives rise to 2^n different vectors. In an AV representation each feature (i.e. *attribute*) can take on values of arbitrary domains. For example, if the robot's sensors can take on values from the domain $0 \dots 10$ depending on the light intensity, a concrete situation can be modeled as $\langle 5, 7 \rangle$, which is sometimes denoted $\{\text{right} = 5, \text{left} = 7\}$. If we denote $\text{ran}(f_i)$ as the *range* of values that feature f_i can take on, the number of different feature vectors of length n can be computed as $\prod_{i=1}^n |\text{ran}(f_i)|$. Note that the range of features can also be a finite set of distinct symbols, instead of numbers. Often, the range of features is continuous, and each feature vector can be seen as a point in \mathbb{R}^n , such that the number of feature vectors is infinite.

From a theoretical viewpoint, propositional and AV representations are almost identical, in that they can be mapped onto each other. A mapping from propositional to AV can be done by mapping true onto 1 and false onto 0. A mapping from AV to propositional can be done by generating a *vector of propositions* for each feature. Let f_i be a feature and

$\{r_1, r_2, \dots, r_k\}$ be its range. The vector of propositions is $\langle f_i = r_1, f_i = r_2, \dots, f_i = r_k \rangle$ and the complete representation is formed by concatenating all vectors for individual features. Propositional languages are a special case of AV representations, where all features are boolean. Note that in case of continuous AV features, the resulting propositional feature vector will be infinite. In the rest of this book we will generally refer to both types of representations as propositional. We will assume that states are represented by feature vectors and actions by discrete symbols.

3.3.3.2 PROPOSITIONAL ABSTRACTION AND GENERALIZATION

Propositional representations allow for many abstraction and generalization mechanisms. Abstractions are usually generated by simple *languages* whereas generalization is often based on *similarity measures*. Many machine learning problems are naturally represented in terms of propositional representations. Usually a feature vector represents an *example*.

A propositional *logical language* Γ consists of a set of *propositions* and a set of *logical connectives* \wedge (and), \vee (or), \neg (not), \rightarrow (implies) and \leftrightarrow (equivalence). The language can be used to express abstractions over propositional feature vectors. For example, the *formula* $\varphi \equiv p_1 \wedge (p_3 \vee p_5)$ abstracts over all vectors in which p_1 is true and either p_3 or p_5 is true. A *ground* interpretation is an assignment of truth values to all propositions. A formula φ *covers* (or, *models*) a ground interpretation i if the truth values of the propositions make the formula true, denoted $i \models \varphi$. Most systems use very simple languages in which only the \wedge -connective is used such that the abstractions can only refer to whether some features are true or false.

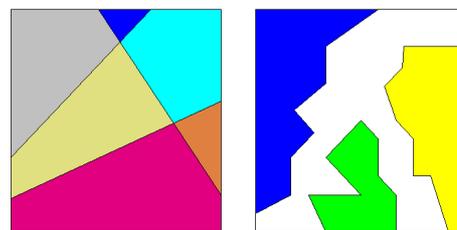


Figure 3.9: *Fences* generated by: **a)** MLP **b)** NN, RBF.

Generalization for AV representations with n features is typically performed by decomposing the n -dimensional space spanned by the feature vectors into regions that group similar vectors. Typical function approximation architectures are based on the fact that input vectors that are close to each other in this space (i.e. in terms of the Euclidean distance) will have similar outputs. Thornton (2000) deals extensively with the way these clusters are formed and used in various learning algorithms and paradigms. In his terminology, most of the learning methods now used are so-called *fence-and-fill* learners. In effect, their approach is to put *fences* (e.g. *hyperplanes*) around regions with some regularity and then they *fill* these regions exhaustively with data points with identically labeled data points. All *similarity-oriented* methods use this *fence-and-fill* learning. Examples of these similarity-oriented, fence-and-fill-learners are *perceptrons*, *nearest-neighbors* (NN) methods, *multi-layer perceptrons* (MLP), *radial basis functions* (RBF), *decision tree algorithms* (DT) and many more. All these methods differ in their ability to form fences, the complexity of the fences used, and the algorithms for learning where fences have to be placed (see more on this in Section 3.6). The common concept is that they use *similarity* as the basis for generalization and that they all use *simple bounding constructs*. This simplicity reflects on the assumption that uniformly labeled data points will tend to exist in relatively simple regions of the input space. Figure 3.9 gives some examples of methods

and the type of fences they can use¹⁹.

3.4. ABSTRACTION TYPE I: State Spaces

State abstraction is one of the most common types of abstraction in MDPs. A suitable transformation (e.g. homomorphism) can map the state space S of the original MDP into an *abstract state space* \mathbf{Z} that is generally much smaller. Sets of states can be treated as a single *abstract state*, by ignoring irrelevant state information. Many types of abstractions such as *factored representations* (see Section 3.5), *local neural networks* and *linear value function approximations* (see Section 3.6) and *hierarchical abstractions* (see Section 3.8) can be casted into a state abstraction framework. In this section, we will describe formalisms and algorithms that *explicitly* deal with state abstractions. All other types of abstraction (Sections 3.5 to 3.8) do use various forms of abstract state spaces, but these are implicitly used in other types of abstraction (e.g. over value functions or policies). We can identify several PIAGET-levels, such as model-minimization (PIAGET-0), exact state aggregations (PIAGET-1) soft state aggregations (PIAGET-2), and several model-free and model-based adaptive resolution methods (PIAGET-3). Their common goal is to build compact representations of the state space. Several methods were developed for continuous environments where abstraction discretizes an originally infinite state space.

In the following we will first describe the basics of state abstraction, focussing on the relation between the criteria based on which states are aggregated and consequences for solutions of the original MDP. We will then go a little more into detail on representations that can be used for state abstractions, and solution algorithms that construct abstract spaces, either in model-based or model-free contexts.

Abstract state spaces. Let us first formalize the concept of state space abstraction. An abstract state space is formed by partitioning an MDP's state space. The state space in Figure 3.5 is such a partitioning of the space in Figure 3.4. Let $M = \langle S, A, T, R \rangle$ be an MDP. An abstract MDP \mathbf{M} is a structure $\langle \mathbf{Z}, A, \mathbf{T}, \mathbf{R} \rangle$. The *abstract state space* $\mathbf{Z} = \{\mathbf{Z}_1, \dots, \mathbf{Z}_n\}$ is a *partition* of the state space S . Typically, $|S| \ll |\mathbf{Z}|$, offering a solution to many of the problems with large state spaces described in the introduction to this chapter. Let ψ denote an *abstraction function* defined as $\psi : S \rightarrow \mathbf{Z}$, which maps each state in the original space S to one of the sets in \mathbf{Z} . In other words, $\psi(s)$ denotes the *membership function*. Now each partition in \mathbf{Z} is defined as $\mathbf{Z}_i = \{s \mid \psi(s) = \mathbf{Z}_i\}$. For all $i, j = 1 \dots n$, \mathbf{Z} satisfies the following properties : i) $\mathbf{Z}_i \subseteq S$, ii) $\bigcup_i \mathbf{Z}_i = S$, and iii) $\mathbf{Z}_i \cap \mathbf{Z}_j = \emptyset$, if $i \neq j$. Since we do not consider action-space abstraction, both M and \mathbf{M} share the same action set A . A transition function and a reward function for \mathbf{M} can now be defined in terms of T and R .

$$\mathbf{R}(\mathbf{Z}_i, a) = \sum_{s \in \mathbf{Z}_i} w(s) \cdot R(s, a) \quad (3.1)$$

$$\mathbf{T}(\mathbf{Z}_i, a, \mathbf{Z}_j) = \sum_{s \in \mathbf{Z}_i} \sum_{s' \in \mathbf{Z}_j} w(s) \cdot T(s, a, s') \quad (3.2)$$

To ensure that \mathbf{T} and \mathbf{R} are well-defined, a weighting function $w : S \rightarrow [0, 1]$ has been added, where for each $\mathbf{Z}_i \in \mathbf{Z}$, $\sum_{s \in \mathbf{Z}_i} w(s) = 1$. The weighting $w(s)$ expresses how much

¹⁹Note that the figure here shows the boundaries for an MLP with hard limited transfer functions instead of the more commonly used sigmoids, which create 'softer' boundaries.

the state s contributes to the abstract state $\mathbf{Z}_I = \psi(s)$. The function w can, for example, be chosen to be in proportion to the state occupancy distribution when an online value learning algorithm is used. An *abstract policy* Π is defined as $\Pi : \mathbf{Z} \rightarrow A$ such that $\pi(s, a) = \Pi(\psi(s), a)$, for all $s \in S, a \in A$. Value functions over the abstract space \mathbf{Z} are denoted $\mathbf{V}^*, \mathbf{V}^\Pi, \mathbf{Q}^*$ and \mathbf{Q}^Π .

An abstract MDP \mathbf{M} comes with a state space \mathbf{Z} for which value functions and policies can be learned as in ground MDPs. For example, Q -learning on an abstract state space can be performed by updating an abstract action-value function based on the (ground) transition (s_t, a_t, r_t, s_{t+1}) as follows:

$$\mathbf{Q}(\psi(s_t), a_t) := \mathbf{Q}(\psi(s_t), a_t) + \alpha \left(r_t + \gamma \cdot \max_{a'} \mathbf{Q}(\psi(s_{t+1}), a') - \mathbf{Q}(\psi(s_t), a_t) \right)$$

In this way, Q -values for abstract states are learned, and these Q -values are shared among all states that are members of the same abstract state. All other types of learning algorithms can be used over abstract spaces, but it depends on *how* states are aggregated (i.e. through ψ) whether convergence can be assured and whether the resulting value function and policy is optimal in the ground MDP. Many authors discuss various properties of state abstraction (e.g. see Singh *et al.*, 1995; Dearden and Boutilier, 1997; Boutilier *et al.*, 1999; Finton, 2002; Givan *et al.*, 2003; Finton, 2005) and numerous others study state abstraction in various algorithms and experimental settings. A unified approach was presented by Li *et al.* (2006) who define five types of state abstraction. These abstractions deal with action-value functions, but similar versions can be defined for state value functions.

1. A **model-irrelevance** abstraction ψ_{model} preserves the one-step model, and an example is the model-minimization approach using bi-simulation by Givan *et al.* (2003). It is defined such that for any action a and any abstract state \mathbf{Z}_i , $\psi_{\text{model}}(s_1) = \psi_{\text{model}}(s_2)$ implies $R(s_1, a) = R(s_2, a)$ and $\sum_{s' \in \mathbf{Z}_i} T(s_1, a, s') = \sum_{s' \in \mathbf{Z}_i} T(s_2, a, s')$.
2. A **\mathbf{Q}^π -irrelevance** abstraction preserves the state-value function for all policies and is defined such that for any policy π and any action a , $\psi_{\mathbf{Q}^\pi}(s_1) = \psi_{\mathbf{Q}^\pi}(s_2)$ implies $Q^\pi(s_1, a) = Q^\pi(s_2, a)$.
3. A **\mathbf{Q}^* -irrelevance** abstraction $\psi_{\mathbf{Q}^*}$ preserves the optimal state-action value function and is related to many value function approximation methods (see Section 3.6) and to exact methods such as structured versions of DP algorithms over factored representations (see Section 3.5). It aggregates states such that for any action a , $\psi_{\mathbf{Q}^*}(s_1) = \psi_{\mathbf{Q}^*}(s_2)$ implies $Q^*(s_1, a) = Q^*(s_2, a)$.
4. An **a^* -irrelevance** abstraction ψ_{a^*} is such that every abstract class has an action a^* that is optimal for all the states in that class and $\psi_{a^*}(s_1) = \psi_{a^*}(s_2)$ implies that $Q^*(s_1, a^*) = \max_a Q^*(s_1, a) = \max_a Q^*(s_2, a) = Q^*(s_2, a^*)$. An example of this type of abstraction is the utile distinction approach (McCallum, 1996).
5. A **π^* -irrelevance** abstraction ψ_{π^*} is such that every abstract class has an action a^* that is optimal for all the states in that class, that is $\psi_{\pi^*}(s_1) = \psi_{\pi^*}(s_2)$ implies that $Q^*(s_1, a^*) = \max_a Q^*(s_1, a)$ and $Q^*(s_2, a^*) = \max_a Q^*(s_2, a)$.

A *partial ordering* relation between abstract state spaces can be defined. Let ψ_1 and ψ_2 be two abstraction functions on a state space S . We can say that ψ_1 is *more fine-grained* than ψ_2 , denoted $\psi_1 \succeq \psi_2$, iff for any states $s_1, s_2 \in S$, $\psi_1(s_1) = \psi_1(s_2)$ implies $\psi_2(s_1) = \psi_2(s_2)$. If $\psi_1 \neq \psi_2$ then ψ_1 is *strictly finer* than ψ_2 , denoted $\psi_1 \succ \psi_2$. The ordering spans a *lattice* with *minimal element* S and as *maximal element* a single abstract state containing all states in S . Each abstraction function induces an *abstraction level*. Comparing different methods for state abstraction can be done by looking at the abstraction function ψ . The ordering among different types of abstraction is the following (Li *et al.*, 2006, Theorem 2):

$$\psi_0 \succeq \psi_{\text{model}} \succeq \psi_{Q^\pi} \succeq \psi_{Q^*} \succeq \psi_{a^*} \succeq \psi_{\pi^*} \quad (3.3)$$

Here ψ_0 is the identity function, i.e. in which no state abstraction is used. Various algorithms employ different kinds of state abstraction and a choice for a particular type introduces a tradeoff between minimizing the amount of information that is lost in the abstraction and maximizing the reduction of the state space size. When the abstract state space is coarser, the amount of information that is lost about the original MDP is larger, and the convergence and optimality guarantees become weaker. When model-based algorithms are used over the abstract space, the resulting abstract policy Π is optimal in the ground MDP for the first four types of abstraction, whereas for a model-free algorithm such as Q -learning, the resulting optimal value function (and the derived policy) is optimal in the ground MDP for only the first three types of abstraction (Li *et al.*, 2006, Theorems 3 and 4). Model-building RL algorithms are guaranteed to converge to the optimal abstract value function whose greedy policy is optimal in the ground MDP, if the weighting function is fixed. We will return to this topic in Chapter 5. The partial observability (see Section 2.7) introduced by the abstractions is the main cause for the lack of guarantees in the fourth and fifth type, and convergence can only be ensured when a fixed policy is used (guaranteeing a stationary state occupation probability).

In general, for most types of state aggregations, one can prove that value learning algorithms will converge, although the resulting policy might not be optimal in the ground MDP (see also Singh *et al.*, 1995). Value learning will converge as long as the value function representation is an *averager* (Gordon, 1995b,a) or the learning algorithm uses an *averaging update rule* (Wiering, 2004). In general, as long as the value function approximation just *interpolates* between values, convergence can be assured (Szepesvari and Smart, 2004). Exact state aggregations are interpolators such that convergence on an abstract level can be proved, though it depends on the aggregation criteria (see the beginning of this section) whether the resulting policy will be optimal in the flat MDP.

In addition to exact aggregations, i.e. in which the abstract state space forms an exhaustive, exact partition of the original state space, a somewhat more general definition is based on *soft state aggregation* (Singh *et al.*, 1995). Let \mathbf{Z} be the abstract state space based on S . Now ψ defines a *cluster probability* as $\psi : S \times \mathbf{Z} \rightarrow [0, 1]$ such that for all $s \in S$: $\sum_{\mathbf{Z}_i \in \mathbf{Z}} \psi(s, \mathbf{Z}_i) = 1$, i.e. each state $s \in S$ can belong partially to several abstract states, i.e. *clusters*. The *value* of a state now generalizes to all states in proportion to the clustering probabilities. Note that exact aggregations are a special case of soft aggregations, in which each state $s \in S$ belongs to exactly one abstract state $\mathbf{Z}_i \in \mathbf{Z}$. Singh *et al.* (1995) prove that for this more general model Q -learning converges to a set of Bellman equations using transition and reward functions as in Equations 3.1 and 3.2, where $w(s)$ is equal to the probability of being in state s under the current policy. The policy is required to be

persistently exciting, under which the Markov chain is ergodic. Further approximations are possible by deliberately violating some of the exact properties in any of the abstractions. In this way, coarser state spaces are obtained at the price of possibly less-than-optimal performance. We will return to this topic in Section 3.6.

So far, we have considered *state abstractions*, i.e. the action set is left unchanged. Little work has been done on *explicit action abstraction* in the MDP setting. The work by Doan and Haddawy (1995), Dean *et al.* (1998) and Ravindran and Barto (2003a) are notable exceptions in this respect. Other work on action abstraction is either implicit (e.g. through the use of *function approximation* in the joint state-action space, see Section 3.6) or based on *temporal action patterns* such as in *hierarchical RL* (see Section 3.8). Starting from Chapter 4 we will see that in the *relational* setting, explicit action abstraction becomes an important issue due to the fact that actions share parameters with state abstractions, such that most abstractions are situated in the joint state-action space.

Learning and Representations. The formal definitions of state space abstractions do not consider *how* these aggregations are being represented. Representing and manipulating them as *sets* is generally not feasible. Again, knowledge representation schemes can aid here in compactly describing the aggregates. In the previous paragraphs we have defined the membership function ψ which determines which states are aggregated as sets. In the propositional setting, there are several ways of implementing ψ . If we assume a featural representation, one can employ all methods discussed in Section 3.3.3.2 for abstraction and generalization. A general way to see the abstract states is to view them as *descriptions* in some KR language, where ψ can act as either \vdash when the language is propositional logic, or any kind of similarity (or distance) measure such as the Euclidean or Manhattan distance. An alternative is to work with *prototype* states that are points in the input space that represent *cluster centers*. In this case, ψ is expressed using a distance measure or kernel and either an input vector belongs to the closest prototype, or it belongs to several prototypes by means of soft state aggregation, similar to k -nearest neighbor approaches.

If a model of the MDP is available, a suitable aggregation can be computed exactly. Without a model, deciding which states are aggregated must be based on statistical evidence gathered through experience, and often *local* tests and splitting operators are used in iterative algorithms that mix experience generation with manipulation of the aggregations (PIAGET-3). A useful criterium is for example that the states currently aggregated in some abstract state display too much variance in their values (Chapman and Kaelbling, 1991). Another way is to look at transition probabilities or the current policy to find useful splits of abstract states that help a useful grouping of states (either on policy distinctions or values, e.g. see Hengst, 2002; Finton, 2005).

Often a *factored representation* is used (see also Section 3.5) which – for the purpose of this section – can be seen as a simple propositional formalism. In essence, various kinds of abstraction on *featural representations* can be seen forming aggregate states, when abstract states are represented by partial assignments of values to features. Other types of representations often used are *decision trees* (see Section 3.6) and KD-trees, which are generalizations of binary trees. In a KD-tree the *root* node represents the entire input space, each branch splits the parent region into one of two discrete sub-spaces along a single dimension, and only the leaf nodes contain actual data about their particular sub-space (e.g. a value). KD-trees are especially useful for *continuous* input spaces, (but see Santamaria *et al.*, 1997; Smart and Kaelbling, 2000; Munos and Moore, 2002; Feng *et al.*,

2004; Roy and Thrun, 2005, for related approaches). Representations for continuous input spaces include *manifolds* and *local neural network* architectures (see Section 3.6), often called *adaptive state space quantization* techniques, and in which state aggregations are often related to topological distances and structures in the input space.

Several methods focus on the *structure* of the input space to define useful aggregations. Standard generalization and aggregation techniques rely on *Euclidean distance measures* to group states, i.e. states that are *close* to each other in the input space taken together. By assuming a different *topological* structure of the state space and by defining different *distance measures* many new ways of grouping states become possible, especially in continuous input spaces. Ferns *et al.* (2004) defines new *metrics* (or, *distance functions*) on states, based on the notion of *bi-simulation* (Givan *et al.*, 2003) described in the next sections. The defined metrics are more robust than *equivalence relations*, such that when two bi-similar states are slightly perturbed, they are still considered *close* in the input space. The use of *manifolds* (which are very related to CMAC, *Kanerva* and SDM codings, see Section 3.6) was considered for navigational domains by Smart (2004) and Glaubius *et al.* (2005). Manifolds are low-dimensional representations of higher-dimensional spaces, on which certain structures can be imposed that are more suitable for the learning task. Related ideas for imposing various *topologies* in spatial, navigational domains were developed by Lane and Wilson (2005). Learned policies in this approach are *relocatable* through the topological abstraction. The intuitive idea in all these methods is that *value function approximation* can be performed more easily by changing the structure of the state space, such that states that are close in the new space will have similar MDP-related properties such as transition probabilities and value.

3.4.1 Model-Based State Abstractions

Models can be used build state space aggregations, and in addition local criteria based on policy distinctions and value function approximations can be used. Some theoretical results by (Even-Dar and Mansour, 2003) show that even for the tabular case, finding a minimal ϵ -equivalent MDP is NP-hard (but deciding whether two MDPs are equivalent can be computed in polynomial time).

Aggregation in Dynamic Programming. Aggregation has been studied in many DP approaches (e.g. see Bertsekas and Castañon, 1989; Baum and Nicholson, 1998; Zhang and Baras, 2001; Lambert III *et al.*, 2004, and accompanying references). Much of this work uses aggregation and disaggregation techniques such that existing iterative procedures are accelerated. For example, Bertsekas and Castañon (1989)'s method groups states in between runs of value iteration, based on Bellman residuals, whereas the work by Baum and Nicholson (1998) forms *non-uniform*²⁰ abstractions over the state space based on *envelopes* (see also Chapter 2). This algorithm uses trees as compact representations of the transition function and performs abstraction by ignoring state features for representing the envelopes. Two other early methods that use KD-trees for representing the state space aggregation, with varying granularity in different regions, are the *parti-game* algorithm and VRDP. The PARTI-GAME algorithm (Moore and Atkeson, 1995) is an adaptive resolution method for finding trajectories to goal regions in high-dimensional continuous spaces by partitioning the input space dynamically into *hyper-rectangles* of varying sizes, repre-

²⁰i.e. where the granularity of the abstraction varies throughout the space.

sented by a KD-tree. Instead of creating state aggregations of similar size over the entire state space, the algorithm starts with a single large aggregation, and splits it into smaller ones in regions of the state space which require fine-grained discrimination to produce a good policy. The algorithm assumes that a *local controller* is available that can take the agent from the current state to a specific state. If taking an action always results in the same transition to a new aggregation, there is no need to split the source aggregation – an effective policy choice for that area of state space has been identified. However, if an occasion arises where an unexpected transition occurs and the action fails to make good progress towards the goal, there must be a state within the aggregation which requires a different action choice, so the aggregation is split. PARTI-GAME assumes that the task is deterministic, there is a known region of the space which is the goal, and the actions consist of moving to neighboring regions in the space. PARTI-GAME treats states where the agent’s progress is blocked as the important states. This is compatible with a definition of importance in terms of differences of values and policies between states; the agent’s failure at these states would typically lead to negative feedback, while some other choice of action would result in success and positive feedback. Several robust extensions to PARTI-GAME were proposed by Al-Ansari and Williams (1999).

Variable resolution dynamic programming (VRDP) (Moore, 1991) too partitions the state space into boxes, indexed by a KD-tree. It learns a model of the environment, which it uses to conduct mental practice runs. States visited during mental practice are considered important, and their boxes receive the finest level of partitioning. The result is a representation with a fine resolution along trajectories through the state space that correspond to the mental practice. VRDP assumes that all states along the trajectory are of equal importance, and ought to be represented at the highest resolution. This assumption does not always hold, and may lead to irrelevant distinctions in control tasks. VRDP also requires the agent to make a good initial guess at a valid trajectory, which prevents the technique from being useful for the general RL problem, in which the agent has no way of knowing whether there is a goal state, or where it might be. Building the model of the environment is entirely separated from the computation of an optimal policy. For control tasks this can be a disadvantage, i.e. as it is blind to the reinforcements, it is bound to create many distinctions that are irrelevant to choosing the correct action.

Yet another, but very related, class of algorithms was introduced by Munos and Moore (1999, 2002) who evaluated the performance of several different splitting criteria in adaptive resolution methods, developed for continuous time and space, deterministic control problems. Again, the discretization is represented in terms of KD-trees. Local splitting criteria based on both the value function approximation and the policy are studied, as well as a global criterion.

Constructing Abstract State Spaces. Whereas the previous approaches do state aggregation while learning, the *model-minimization* approach by Givan *et al.* (2003) (see also Dean and Givan, 1997) computes a compact representation directly from the original (factored) MDP representation. The minimized MDP can then be solved using classical DP techniques. The approach takes inspiration from minimization algorithms for *finite state automata* (FSA) and in addition from the connection between these algorithms and the use of *regression* in STRIPS planning (see Givan and Dean, 1997). The key contribution is an extension of the concept of *bi-simulation* from the FSA literature to *stochastic bi-simulations* for MDPs, incorporating stochastic transitions and rewards. Stochastic bi-

simulations implement a *model-irrelevance* abstraction (see previous section) in which the one-step transition probabilities and immediate rewards of abstract states are used as a criterion for deciding whether to aggregate states. The algorithm starts with a single abstract state and then iteratively refines the partitioning if a *block* in the partition violates the aggregation criterion, i.e. it is *unstable*. By repeatedly finding unstable blocks and splitting them, the target partition can be found in linearly many splits in the target partition size. Givan *et al.* (2003) define a number of ways to implement the splitting criteria and show that related algorithms for solving *factored* MDPs (see Section 3.5) can be cast into the model-minimization formalism. A difference is that algorithms such as SVI and SPI (see Boutilier *et al.*, 2000a) split aggregate states based on the value function as well, such that they do model-minimization and value learning in parallel.

Several extensions in the context of factored representations concern the use of minimization in the context of both large state and action spaces (Dean *et al.*, 1998), in which the abstractions range over the joint state-action space. Furthermore, as an extension of *exact* model-minimization, Dean *et al.* (1997) study *bounded parameter* MDPs, in which transition probabilities do not have to be *exactly* the same for states within the same block, but instead lower and upper bounds are taken for transition probabilities and rewards. Kim and Dean (2003) describe a related algorithm for factored MDPs in which *non-homogenous*²¹ blocks are allowed. The algorithm constructs an approximate aggregate MDP, solves it exactly and uses the resulting value function as a heuristic to refine the current best non-homogenous partition. Both algorithms relax the constraint that the abstract MDP should be equivalent to the original MDP and instead allow the abstract model to be *similar* to the original, presumably yielding a more compact abstract model.

Ravindran (2004) (see also Ravindran and Barto, 2001) takes an *algebraic* approach and describes a model-minimization approach for MDP that is based on *homomorphisms* (see also (Korf, 1980) and Section 3.2). Similar to the approach by Givan *et al.* (2003) it is based on the FSA literature. However, it extends Givan *et al.* (2003)'s approach by considering *state-action equivalences*, which enables to exploit *symmetries* in the MDP which is not possible in the bi-simulation approach. A related approach explicitly dealing with symmetries too was proposed by Zinkevich and Balch (2001). The homomorphisms approach was later extended to *semi*-MDPs to compute minimized models in the context of temporal abstraction (Ravindran and Barto, 2003a, 2002, 2003b), but see Section 3.8 for a more extensive description.

Dearden and Boutilier (1997) (see also Boutilier and Dearden, 1994) use *probabilistic* STRIPS rules (see Section 3.5) to *intensionally* represent an MDP. They use information derived from the factored representation to rank the state variables according to their degree of influence on the reward function. Given this ranking, a subset of the most relevant variables can be determined, and an abstract MDP with a smaller state space can be solved using this subset. This solution may then be used to derive a solution for the original MDP, reducing the time to convergence. Dearden and Boutilier also describe a relaxation of their method to derive *inexact* abstractions. Probabilistic STRIPS representations were also used in related algorithms by (Dietterich and Flann, 1995, 1997) (see Section 3.5) and the model-minimization approach in the previous paragraph (see also Givan and Dean, 1997).

²¹i.e. Where abstractions can contain states with different values.

3.4.2 Model-Free State Abstractions

Work that explicitly introduces model-minimization in an online, model-free fashion is usually referred to as *adaptive resolution* techniques. This work is characterized by an iterative procedure in which the abstract representation is modified in parallel with value learning methods that drive the representation modification process. This can also be seen as a modified (or even generalized) policy iteration process in which the abstract representation of the value function and policy can change constantly, as in PIAGET-3. Sampled state-action pairs or solution traces are used to build compact models of the underlying MDP. Model-free RL can be employed over abstract state spaces (e.g. Singh *et al.*, 1995) but several methods have tried to do this adaptively. Most model-free methods iteratively refine the abstract state space by *splitting* abstract states based on the current value function approximation.

The *multi-resolution* method KD- Q -learning (Vollbrecht, 1999) starts with a fully partitioned state space, up to a certain resolution. Q -values are stored and learned on *each level*. Although a disadvantage is that the number of Q -values is still very large, the algorithm can select Q -values from different resolutions, whichever one has the highest confidence. An added benefit of having multiple levels of Q -values is that more effective use is made of experience. Related to this method is FEUDAL Q -learning (Dayan and Hinton, 1993; Dayan, 1998) in which multiple resolutions are represented in a hierarchical structure, though in FEUDAL, the policy is hierarchically structured too (see more in Section 3.8).

The *adaptive resolution method* by Reynolds (2002, 1999, 2000)'s *decision boundary partitioning* algorithm is based on KD-trees too. It is an adaptive resolution method that splits abstract states using action value estimates. Regions are split once their action values are sufficiently approximated (e.g. by counting their updates), when *taking the recommended action of one region in the adjacent region is expected to be significantly worse than taking another, better, action in the adjacent region*, which is related to the *loss* the agent experiences when not provided with the right state abstraction.

Finton (2002, 2005) introduces an adaptive resolution algorithm for RL in continuous domains, called the *cognitive economy* framework. State splitting is based on both policy preferences and value functions. It uses an *active state investigation* strategy to find possible blocks to be split. Finton (2002) includes an extensive analysis of the theoretical properties of the criteria for state aggregations.

Many algorithms in this chapter can be seen as performing adaptive resolution. Specifically mentioned often in these contexts are the UTREE (McCallum, 1995; McCallum, 1996) and G-algorithm (Chapman and Kaelbling, 1991) which will be described in Section 3.6 and adaptive state space quantization using local neural cell structures (Großmann, 2000) which can be found among the neural network descriptions in Section 3.6.

3.5. ABSTRACTION TYPE II: Factored Markov Decision Processes

Classical planning uses representations that *intensionally* represent the problem, i.e. in which *descriptions* of things are in terms of properties they have. Probabilistic approaches in AI have introduced structural representations of probability distributions in terms of *graphical models* such as Bayesian networks. Such intensional descriptions and graphical models can be extended to MDPs. *Factored representations* (Boutilier *et al.*, 1999; Boutilier, 1999) are compact, structured representations of the MDP, policies and value

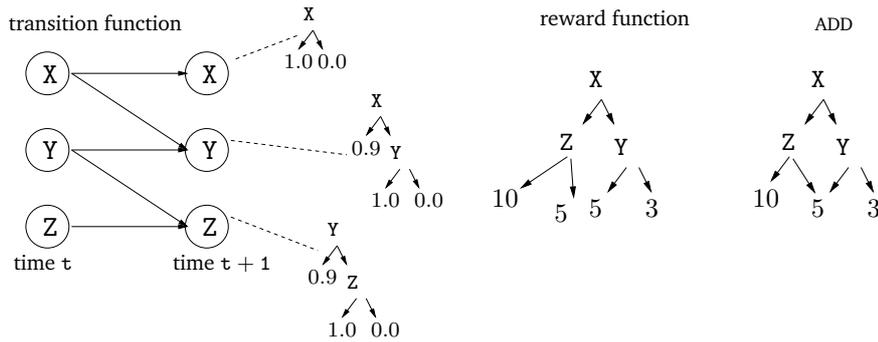


Figure 3.10: An example of a *factored representation*. **a)** A 2-time slice dynamic Bayesian network (DBN) representation of the probabilistic dependencies between state variables between time steps, for a given action. The conditional probability tables are replaced by compact tree-based representations. **b)** A compact representation of the state reward function. **c)** An algebraic decision diagram in which subtrees are shared.

functions. They allow for *structured versions* of classical MDP solution techniques such as policy and value iteration. A key element in all these approaches are *structured representations* such as *rules*, *graphs* and *trees* in combination with efficient operations on these structures. Because these structures *intensionally* represent *state aggregations*, policies and value functions, solution techniques work on the level of abstractions, *without enumerating the complete state space*. Because new aggregations are constantly formed during computation, by manipulating structured representations, most of these algorithms belong to PIAGET-3, computing structures and parameters in parallel. Several algorithms based on *linear approximations* of value functions belong to the PIAGET-2 class, because they fix the structural descriptions and focus on parameter estimation.

3.5.1 Structured Representation

Once it is assumed that states can be factored into *features*, compact representations of the common structures such as the transition function and policies can be given by exploiting *regularities* and *independencies* between features.

Factored Dynamics. One of the problems of huge MDPs is the size of the transition matrix. For a state containing 10 binary features $F_1 \dots F_{10}$, the joint action probability for action a is $P(F'_1, \dots, F'_{10} | F_1, \dots, F_{10}, a)$ (where F'_x denotes the value of F_x in the next state) and this amounts to $2^{10} \cdot 2^{10}$ parameters for each action. A compact representational formalism for storing these probabilities are *dynamic Bayesian networks* (DBN) (Dean and Kanawaza, 1989; Ghahramani, 1998). For the DBN in Figure 3.10a we can write the full conditional joint distribution as $P(X_{t+1}, Y_{t+1}, Z_{t+1} | X_t, Y_t, Z_t, a)$. Using the conditional independencies in the DBN we can write this in factored form as:

$$P(X_{t+1}, Y_{t+1}, Z_{t+1} | X_t, Y_t, Z_t, a) = P(X_{t+1} | X_t, a) P(Y_{t+1} | X_t, Y_t, a) P(Z_{t+1} | Y_t, Z_t, a)$$

The *conditional probability table* (CPT) for each node at time $t + 1$ stores the probability that the state variable has a particular value at this time, given the values of the state variables at time t . In the worst case, all variables depend on all other variables, and there is no representational gain. In the best case, all variables are independent such that only

one parameter is needed per (binary) variable. In most practical cases however, variables depend only on a *few* other variables. Arcs in the DBN from time t to $t + 1$ are called *diachronic* whereas arcs from $t + 1$ to $t + 1$ are called *synchronic*²². Synchronic arcs represent *correlated* action effects and require an additional inference step²³. In addition to the compact representation the DBN provides, the CPTs are amenable to this too. Figure 3.10a shows how the CPTs for each node at time $t + 1$ can be represented using a decision tree, although one can use more compact *decision graphs* as well.

An alternative for the factored representation of action dynamics is the PSTRIPS (Hanks and McDermott, 1994) formalism (e.g. see Boutilier and Dearden, 1994; Dearden and Boutilier, 1997). This is a probabilistic version of the well-known STRIPS (Fikes and Nilsson, 1971) operators in which the effects of an action are represented by *add* and *delete* lists. The probabilistic extension adds the ability of an action having multiple, probabilistic *outcomes*. For example, a PSTRIPS *operator* (PSO) with n outcomes might be specified as

$$\langle \text{pre}, \{(\text{ADD}_1, \text{DEL}_1, p_1), \dots, (\text{ADD}_n, \text{DEL}_n, p_n)\} \rangle$$

where *pre* is the condition under which the rule can be applied (i.e. an action precondition), and ADD_i and DEL_i are the add and delete lists for the i -th outcome, and the probabilities p_i of all outcomes i sum up to one. A concrete rule is $\langle [X, Y] \{ ([Z], [X], 0.3), ([], [X, Y], 0.7) \} \rangle$ in which after applying the rule on the state $X \wedge Y \wedge \neg Z$ will imply a transition to state $\neg X \wedge \neg Y \wedge \neg Z$ with probability 0.7. The PSOs can be more compactly represented using a (binary) decision tree formalism. Note that DBNs model influence on individual features, whereas PSO explicitly represent multiple effects of an action, such that PSOs are more convenient when many effects of the action are correlated, i.e. the correlated effects are *compiled* into the operators. PSOs and DBNs can be proved representationally equivalent (Littman, 1997). PSOs can be more favorable when most of the action's effects are correlated, although in many cases the DBN representation is exponentially smaller than the equivalent PSO. Theoretical results by Allender *et al.* (2002) show that DBNs are not guaranteed to be compact, but in practice they often are.

Factored Reward Functions, Value Functions and Policies. Like an action's effect on a particular feature, the reward associated with a state often depends only on a subset of all features. This fact can be used to represent reward functions in structured form, for example using a decision tree, see Figure 3.10b. Factored states, actions and transition and reward functions are compact representations of the specification of the MDP itself. The solution parts, i.e. value functions and policies, are also candidates for structured representation. Trees and graphs, but also *rules* can be used to exploit structure and *aggregate* states that have similar value or the same policy action. Decision trees have been used in other work for representing the value function, see Section 3.6.2.3. Trees can be built using tests on state variables. Leaves of a *policy tree* (resp. *value tree*) are labeled with an action (resp. a real value), denoting the action (resp. the value) of all states consistent with the labeling of the corresponding branch. Some advanced algorithms (Hoey *et al.*,

²²Note that Bayesian networks, including DBNs, have to be acyclic, such that not all synchronic nodes can be present because they would introduce a cycle.

²³An alternative is to eliminate synchronic effects by transforming the model, with a possible blow-up of the model (see Boutilier *et al.*, 1999, Section 4.2.6.).

1999; St-Aubin *et al.*, 2001) use highly compact *algebraic decision diagrams* (ADD) instead of trees. ADDs are an extension of *Boolean decision diagrams* that allow terminal nodes to be labeled with real values instead of just boolean values. ADDs are similar to trees, except that ADDs allow for sharing of *isomorphic subtrees*, i.e. arcs of the tree can lead from one branch to another such that shared parts are represented only once, see Figure 3.10c for an example.

3.5.2 Structured Algorithms

Factored representations can be used as the basis for *structured versions* of policy and value iteration (see Dearden, 2000; Boutilier *et al.*, 2000a, for extensive overviews). The basic assumptions in all these approaches are:

- If we have a structured version of π and a structured estimate of V_t of the value of π , we can compute an improved estimate V_{t+1} that often preserves much structure
- If we have a structured estimate of V^π of the current policy, we can compute an improving policy π' while maintaining the structured representation.

The core of all approaches is to translate operations that are part of standard DP algorithms – such as Bellman backups, maximization over Q -values to compute V -values, and arg-maximization over V -values to compute policies – to equivalent, efficient manipulations of structured representations such as trees, ADDs and rules. In this way, compact, intensional descriptions are used instead of the flat MDP such that these algorithms can deal with huge MDPs, exploiting their structure to avoid explicit state space enumeration..

Here we will briefly describe a number of approaches, though in this section we will not describe the algorithms in full detail. In Chapter 6 we will introduce *set-based dynamic programming* and *intensional dynamic programming* which can be seen as a general mechanism underlying most structured versions of value and policy iteration, with possible implementations using factored representations, continuous MDPs and relational formalisms. There we will introduce a new structured value iteration algorithm for *relational* domains called REBEL and discuss relations with other algorithms using structured representations.

Model-Based Algorithms. Techniques such as *policy iteration* and *value iteration* can be performed over the tree structure of policies and value functions, which means that computation is focused on the necessary parts of the state space, *without explicit state enumeration*. The key to all these algorithms is a *decision-theoretic regression operator* (DTR) (e.g. see Boutilier *et al.*, 1999) used to construct the Q -functions for an action a given a particular value function V . If the value function is tree-structured, this algorithm produces a tree-structured representation of the Q -function, i.e. a Q -tree. DTR is a probabilistic extension of *goal regression* as used in classical planning systems (see Russell and Norvig, 2003). There, the regression of a set C of conditions through an action a is the *weakest set of preconditions such that performing a will make C true*.

DTR is a probabilistic analogue to this process. Because we are concerned with values rather than regressing a single condition, regression is performed on a set of conditions, each with an associated value, through an action. And because the actions have probabilistic effects, there is no absolute set of preconditions for making any condition C true.

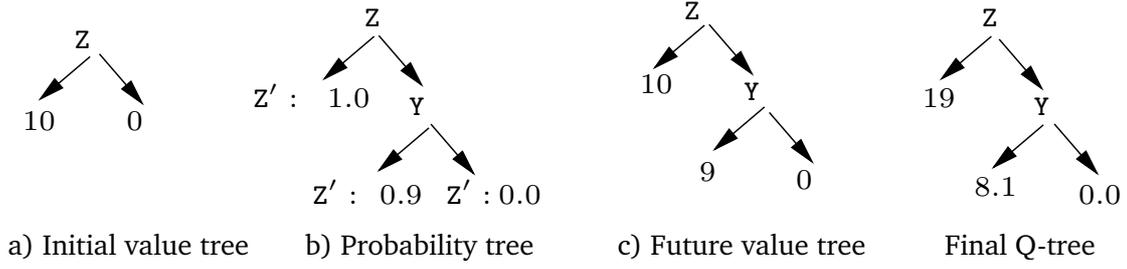


Figure 3.11: DTR of $\text{Tree}(V)$ through action a in Figure 3.10 to produce $\text{Tree}(Q_a^V)$: **a)** $\text{Tree}(V) = \text{Tree}(R)$, **b)** $\text{PTree}(V, a)$, **c)** $\text{FVTree}(V, a)$, **d)** $\text{Tree}(Q_a^V)$.

Instead, there exists sets of preconditions under which action a will make each of the regressed conditions true with identical probability. Since each of the conditions has a value associated with it, decision-theoretic regression produces sets of preconditions each of which has the same *expected value* under a .

Let a be an action, and let V be the value function represented by a tree. To produce the Q -function Q_a based on V , we need to determine the probabilities with which different states s make the conditions indicated by the branches of V true. To compute the Q -function $\text{Tree}(Q_a^V)$ with respect to $\text{Tree}(V)$ we perform decision-theoretic regression through a single action a . We use the structure in the reward function, the current value function and the representation of a to determine when states have the same Q -values and can be grouped together. Two states s_i and s_j have the same Q -value if they have the same reward and the same expected future value. Their rewards can be simply read of the $\text{Tree}(R)$ such that we focus on their future value. Several operations are needed to perform DTR on trees, such as *simplification*, *appending* and *merging*. These operations implicitly manipulate and combine state space aggregations.

As an example, consider the DBN in Figure 3.10. A reward tree $\text{Tree}(R)$ is depicted in Figure 3.11a. Let the initial $\text{Tree}(V)$ be $\text{Tree}(R)$. Figure 3.10 depicts the steps in computing $\text{Tree}(Q_a^V)$. V depends only on Z such that the expected future value only depends on the conditions that affect the probability of Z being true after applying a , and these conditions can be found in the CPT for Z . This tells us that Z depends on the values of Y and Z in the previous state, shown in the probability tree in Figure 3.11b. At this point it is straightforward to compute the expected future value tree for action a , shown in Figure 3.11c. For example, when Z is false and Y is true before the action, Z becomes true with probability 0.9 after the action, and this state has value 10 and Z remains false with probability 0.1, which has value 0. So, the expected future value of all states where these conditions hold is 9. The $\text{FVTree}(V, a)$ is the future value tree, which is obtained by converting the distributions at each leaf of $\text{PTree}(V, a)$ to expected values w.r.t. V . Finally, Figure 3.11d shows the Q -tree $\text{Tree}(Q_a^V)$ which is obtained by applying the discount factor at each leaf of $\text{FVTree}(V, a)$ and then merging this tree with the reward tree $\text{Tree}(R)$, summing the values in the leaves. Because $\text{Tree}(R)$ only contains Z , the structure of the tree is unchanged from that of $\text{FVTree}(V, a)$. Note that if the trees (especially the value tree) contains richer structure, these operations become more complex.

DTR forms the basis for a number of structured DP methods for factored MDPs. Boutilier *et al.* (1995) devised the first algorithm for *structured policy iteration* (SPI). Its general structure can be stated as follows:

- i) Choose a random *structured policy* π , then loop through the following two steps
- ii) Approximate the value function V^π using *structured successive approximation*
- iii) Produce an improved structured policy π' (if no improvement possible; terminate)

Structured successive approximation (SSA, see Step ii) applies several complete iterations of *full backups* of DTR through the policy to compute the value tree $\text{Tree}(V^\pi)$. Related to SPI, the *structured value iteration* (SVI) algorithm (Boutilier and Dearden, 1996) employs similar operations, now performed over *all* actions combined with a *maximization* operation to find a tree maximizing the values for all states, given the current value function estimate V_t . This approach was extended to *approximate* SVI by limiting the tree size such that states differing only slightly in value are aggregated together in $\text{Tree}(V_t)$, i.e. leaves represent *possible ranges of values*. Boutilier (1997) describes how to deal with the more difficult problem of *correlated effects* (i.e. *synchronic arcs*) in the DBN action specification, which requires some extra technical machinery and probability computations not present in the original SPI and SVI algorithms. The SPUDD algorithm (Hoey *et al.*, 1999, 2000) is similar to SVI, but it replaces trees by ADDs. The conceptual structure is unaffected by this, but due to the more compact ADD representations (and more efficient algorithms for manipulating them) SPUDD is much faster than SVI. In fact, Boutilier *et al.* (2000a, p. 103) note that *'the same principles apply to any structured representation as long as one can develop a suitable regression operator for that representation.* Evidence supporting this claim are a similar method for *continuous* MDPs developed by Feng *et al.* (2004) and DP approaches for *relational* representations which can be found in Chapter 6. SPUDD was generalized to an approximate version APRICODD (St-Aubin *et al.*, 2001) along the same lines in which SVI was generalized to approximate SVI. The *symbolic* RTDP algorithm (Feng *et al.*, 2003) and the *symbolic* LAO* (Feng and Hansen, 2002) are extensions of RTDP and LAO* (see Section 2.5.2.2) to factored MDPs.

The structured algorithms SPI and SVI (and in addition the ADD-based algorithms and corresponding approximate versions) can be seen as implementing *model-minimization* (MM) of a factored MDP representation. Remember from Section 3.4.1 that these algorithms first compute a reduced model before learning value functions and policies for this reduced model. Because SPI and SVI combine both splitting operators and value estimation, their behavior can be seen as "anytime", which can be used in contexts where a full reduction can be too expensive to compute. There are many more connections between the work on structured solution algorithms and MM frameworks (see Givan *et al.*, 2003, for an extensive discussion).

Very much related is the *explanation-based* RL approach (EBRL) by Dietterich and Flann (1997), in which the intimate relationship between Bellman backups and *explanation based learning* (EBL) (Minton *et al.*, 1989) is shown. Like RL (but unlike EBL) it can learn optimal plans. Like EBL, but unlike table-based RL, it can also effectively generalize its experience to other similar states in a manner *justified by the domain theory*. The idea is to use the *inverse* O^{-1} of a PSTRIPS-like operator O to compute a Bellman backup for multiple states, implementing *region-based backups*. The use of EBL-style generalization in TD-learning was already described by Yee *et al.* (1990), who used *region-based* backups in online RL. The system keeps a tree of useful concepts and tries to *generalize* experience to large regions in the input space. Experiments on TIC-TAC-TOE showed that the method outperformed a 6-ply lookahead search algorithm. EBRL was originally developed for deterministic, goal-based grid worlds (Dietterich and Flann, 1995), and was

later extended to stochastic domains. For example, a deterministic action `moveEastToWall` might bring the agent from an arbitrary grid position to the nearest wall in the east direction. In this way, the action maps *multiple* states onto the same grid position s , i.e. the action is said to have the *funnel* property²⁴. By taking the inverse of the action operator, one can perform a Bellman backup of the value of s to update all states that lead to s after applying `moveEastToWall`. These states can be found by *regression* through the action, in a similar way as was done for factored MDPs (see Chapter 6 for examples). In contrast to other generalization techniques such as VFA (see Section 3.6), the generalization in EBRL is *justified* by the model. For *probabilistic* actions, things become more complex. Because a PSTRIPS representation of an action is factored into different *outcomes*, performing full Bellman backups involves the *combination* of the pre-images of the regressed parts. The regression operator for DBN factored dynamics performs this step implicitly because all outcomes of one action are represented in one structure.

The advantage of all these structured approaches is that the structure of value functions and policies is determined *during* the solution of the MDP. There are two problems with *exact*, structured value functions such as SVI's trees and EBRL's *regions*. First, even for trivial problems for which the factored MDP representation is small, the value function representation can be exponentially large (as can the policy). Second, these structured representations are limited to *partitions* of the state space. The number of partitions is *at least* as large as the number of different values. Dietterich and Flann (1997) experimented with rearranging computational steps in their algorithms to derive *coarser* partitions, although the aim is still to represent the exact value function.

One solution to the problem of huge value functions is to limit the size of the trees, allowing states with slightly different values to be mapped onto the same leaf (Boutilier and Dearden, 1996). A similar mechanism was used to limit the size of the representing ADDs in APRICODD (St-Aubin *et al.*, 2001). These solutions can be seen as *approximating the solution*. A related class of algorithms for factored MDPs can be viewed upon as approximating the problem itself, by using *linear* approximations of the value function, instead of trees and ADDs as in the algorithms we discussed previously. Note that linear approximations of the value function is discussed in Section 3.6 where *sampling* using RL is used to find appropriate weights. Here in the model-based setting, the solution can be implemented *in closed form* without enumerating the entire state space. Several approaches use *approximate* LP (as was mentioned in Chapter 2) and *approximate* policy (API) and value iteration (AVI) approaches for factored MDPs. Approximate DP for *factored* MDPs (Koller and Parr, 1999, 2000; Guestrin, 2003; Guestrin *et al.*, 2003b) modify the exact Bellman backup to a backup that *projects* back the value function onto the linear basis function. Approximate LP techniques apply LP to the set of constraints (given by the Bellman equations) on the linear value function approximation (Schuurmans and Patrascu, 2001; Guestrin, 2003; Guestrin *et al.*, 2003b). An important problem for approximate LP algorithms is how to efficiently perform *constraint generation*.

Model-Free Algorithms. Although the SPI and SVI algorithms provide compact representations and efficient structured operations, they still operate over the *complete* value functions. However, there is no reason why the operator cannot be applied more locally on

²⁴Actions having this funnel property show many similarities with *options* (Sutton *et al.*, 1999) and the resulting states can be seen as *subgoals* in hierarchical RL approaches, see Section 3.8. In fact, the EBRL approach was later extended to the hierarchical case by Tadepalli and Dietterich (1997).

smaller parts of the state space. This is the key idea behind *structured prioritized sweeping* (SPS) (Dearden, 2000, 2001) (see Section 2.6.3 for classical PS). Instead of applying the regression operator to the whole value tree, SPS applies it to a *partial assignment* of values to variables, and consequently the *priority queue* is based on these partial assignments instead of single states. The related *generalized prioritized sweeping* by Andre *et al.* (1998) uses similar ideas for efficient updates based on a factored model, though the value function is state-based, in contrast to SPS that uses a structured representation. The EBRL approach by Dietterich and Flann (1997) defines a PS algorithm based on *region-based* Bellman backups, though in an off-line fashion, rendering it more close to asynchronous DP methods. *Online* (RL) versions of EBRL are limited to *deterministic* operators, where the states sampled along a trajectory are enough to perform (full) structured backups.

Factored representations have been used in other model-free of approaches. The factored E^3 (Kearns and Koller, 1999) algorithm is an extension of E^3 (Kearns and Singh, 1998, see also Section 2.6.3). It assumes that the structure of the DBN is given, and in addition, that an approximate planning algorithm is available. It improves upon E^3 in that it achieves near-optimal performance in a running time polynomial in the number of DBN parameters (which is usually exponentially smaller than the number of states). Both Littman *et al.* (2005) and Jonsson and Barto (2005) explore factored representations in *hierarchical* RL (see more in Section 3.8). The use of factored models with DBN action dynamics have also been explored in the context of model-free RL for MDPs and POMDPs (Sallans, 2002; Sallans and Hinton, 2004).

Learning a Factored Model. For most of the methods we have described it is assumed that the factored model is given, or at least the structural part. Several authors have investigated learning this model from data, for example by Dearden (2001) and Jonsson (2006). Two other recent approaches are SDYNA (Degris *et al.*, 2006) and SLF-R-MAX (Strehl *et al.*, 2007). SDYNA is a general approach, but one instantiation is the SPITI algorithm that uses incremental decision tree induction, combined with an incremental version of SVI. The SLF-R-MAX approach uses focused exploration (see R-MAX in Section 2.6.3) and provides formal guarantees on the quality of its (online) behavior. Methods that learn the model structure from data are positioned at PIAGET-3 and carry the promise of being able to learn from scratch, while at the same time learning and exploiting various forms of structure in the environment (as well as policies and value functions).

3.6. ABSTRACTION TYPE III: Value Function Approximation

One – widely studied and validated – way of dealing with large or continuous MDPs is the use of *function approximators* (FA). Instead of explicitly storing values in a lookup-table, a parameterized function can be used. A state value can be computed using $V(s) \equiv F(s, \vec{\theta})$ and the complexity is now primarily related to the dimensionality of $\vec{\theta}$ and not so much to the size of the state space. Function approximation (FA) has been studied in many disciplines, such as inductive concept learning (Alpaydin, 2004), pattern recognition (Bishop, 1995) and statistical curve fitting (Hastie *et al.*, 2001). This section will mainly focus on the *prediction problem*, i.e. learning value functions. However, more elements than just value functions can be approximated. Policies, transition functions and reward functions all have a functional form, such that these can be approximated too, and we will address these issues briefly at the end of this section.

The goal of FA is to develop a computational relationship between *input* and *output* variables based on a *known* dataset of input-output samples (i.e. *examples*). Once this mapping is obtained, it can be used to predict the output for *unseen* inputs. Inputs and outputs can have various forms. Inputs are usually feature vectors, and outputs can be real-valued or discrete symbols. In the context of value function approximation, the output is a single real-valued variable and the problem is known as *regression*. If the outputs are discrete symbols, the problem is called *classification*. In the regression setting, one seeks a function f of n input variables (denoted by vector \vec{x}) from a given dataset of k examples (\vec{x}_i, y_i) , such that $y_i = f(\vec{x}_i)$, $i = 1 \dots k$. A frequently used form of FA is a set of *basis functions* $\varphi_1 \dots \varphi_n$ such that the output y can be computed for input \vec{x} as:

$$y = \tilde{f}(\vec{x}) = \sum_{i=1}^n \theta_i \varphi_i(\vec{\theta}_{\varphi_i}, \vec{x})$$

where the *weights* θ_i and the basis functions' parameters $\vec{\theta}_{\varphi_i}$ have to be learned. The goal is to find a function \tilde{f} such that the "distance" between the function's output and target values in the dataset is minimized. A useful distance is the *mean squared error* (MSE) defined over a dataset of size k :

$$\text{MSE} = \frac{1}{k} \sum_{i=1}^k \left(y_i - \tilde{f}(\vec{x}_i) \right)^2$$

This MSE criterion has convenient mathematical properties and is used often (see Hastie *et al.*, 2001). Note that there is no unique solution to this minimization problem. Given a dataset of size k there is an infinite number of functions that interpolate these k examples with a minimized MSE. In order to obtain a suitable approximation, one has to restrict the space of functions that is considered using a particular *bias*. In learning problems this bias – or, *model selection* – consists of the choice for a particular *approximation architecture* and a *training method*. The architecture bias limits the space of functions, and the training method bias determines how this restricted space is searched. The table-based representations of the previous chapter are, in that respect, completely *unbiased*.

An important dimension of function approximation is the fact that the function is learned from a *limited* dataset that typically contains only a fraction of the entire input range. It is important that the *distribution* of samples in the dataset resembles the true distribution of the problem, such that the learned mapping will apply to unseen cases in an appropriate way. Related to this is how *accurately* the approximation architecture can *represent* this mapping. Highly complex architectures can represent arbitrary complex functions. However, given the limited number of examples, it is often preferable to have a simpler architectures that represent functions that are more *smooth*, to prevent *overfitting* of the data. Overfitting happens when the data is fit perfectly using a overly complex mapping that exactly models the input-output behavior, but performs poorly on unseen cases. In general there is a trade-off between the complexity of the function class, the amount of training data and the generalization error on new examples. A useful taxonomy of FAs distinguishes between *local* or *global*, and *static* or *adaptive* architectures.

Local versus Global Architectures. One important distinction between function approximation architectures is their *degree of locality*. An FA is called *local* if it uses only a small

part of its processing elements for computing an output given an input. The *table* representations of the previous chapter are extreme cases of local models because they use only one element (i.e. the table entry). Examples of local models are various forms of local neural networks such as *Kohonen networks*, *neural gas* and *neural cell structures* (see (Haykin, 1999) and Section 3.6.2.2), *decision trees* (see Section 3.6.2.3) and CMACs (see next sections). A *global* FA on the other hand, uses most or all of its processing elements to compute an output for a given input. The interpolation between training examples is usually more smooth, but due to the *interaction* (or, *interference*) between the FA's parameters for different inputs, they tend to *overgeneralize* more and they are more vulnerable to *forgetting* (French, 1999). Forgetting happens when the FA is trained on some examples in one part of the input space, it tries to optimize tune its parameters towards this part, but thereby forgetting what it had learned before for other parts of the space. Local models are less affected by this, e.g. some parts of the model can be adapted without affecting the approximation in other parts of the input space. A possible disadvantage is that they are less capable of grasping global characteristics of the input space. Usually, local models need to be much larger because of this. Examples of global FAs are *multi-layer perceptrons* (Bishop, 1995) and *linear networks* (see next section).

Static versus Adaptive Architectures. In a *static* architecture, the basis functions φ_i are *given* and training the architecture corresponds to finding good parameters θ_i . In *adaptive* architectures, the basis functions φ_i are subject to change as well, i.e. the set of parameters θ_{φ_i} is adapted too, or extended. Adaptive architectures are more flexible, but also more difficult to train. The advantage is that there is no need to decide a priori on the right bias, i.e. the structure and parameters, and that the model adapts to the data during learning. Within the class of adaptive architectures, we can further distinguish between architectures that have a fixed number of basis functions or processing elements, and architectures that can be *extended* during learning, e.g. by adding processing elements when needed. For example, in some local FAs, such as RBF networks, one can move the centers of the RBFs around, but one can also add additional RBFs in areas where the current approximation is still poor (see next section). A special case are the *adaptive resolution* algorithms (see Section 3.4.2) that increase the resolution (of e.g. the state space representation) online.

Training. *Training* a *static* architecture consists of finding the appropriate setting of the architecture's parameters $\vec{\theta}$. The particular way of doing this depends on the type of architecture. Often $\vec{\theta}$ is a real-valued vector in n dimensions and one can devise an algorithm that travels through \mathbb{R}^n and picks the point in space that minimizes some cost function $C : \mathbb{R}^n \rightarrow \mathbb{R}$, for example MSE. Many algorithms use the *gradient* of C with respect to the parameters $\vec{\theta}$ as a *direction* for this travel. Examples of these so-called *hill-climbing methods* are *steepest descent* and *Newton methods* (see Bishop, 1995). In *batch* (or *off-line*) mode training, this gradient is computed using the dataset, and the parameter vector is moved along the gradient direction. In *online* training methods, examples are presented one-by-one and the parameter vector is changed after each example. In both training methods, a *learning parameter*, either fixed or diminishing, determines the *step-size*, or the rate of change. Online training methods are useful for large training sets, and especially in (model-free) RL where training examples become available one at a time during interaction with the environment.

Training an *adaptive* architecture consists of two learning problems; learning the pa-

parameters as in static architectures (i.e. *parameter estimation*) and, in addition, learning the basis functions (i.e. *structure adaptation*). Usually these two problems are approached by interleaving both. First, an initial architecture is chosen and its parameters are trained. Based on statistical estimates concerning error or variance, parts of the architecture are adapted, or the architecture is extended with new basis functions. The general form of this mechanism is similar in spirit to *structural* versions (Friedman, 1998) of the *Expectation-Maximization* (EM) algorithm (Dempster *et al.*, 1977), a general framework for problems in which both parameters and structure are learned, for example in *probabilistic graphical models* such as *Bayesian networks* and *hidden Markov models* (see Jordan, 1999)

3.6.1 Fundamentals of Value Function Approximation

Function approximation can be used for approximating value functions in RL (Bertsekas and Tsitsiklis, 1996; Keerthi and Ravindran, 1997, Chapter C3.6). In fact, one of the earliest works, that of learning CHECKERS by Samuel (1959), used a linear combination of numerical features as evaluation function. Consider an algorithm such as Q -learning. Normally, the value $Q(s, a)$ is given by the value in a table representing Q . When *value function approximation* (VFA) is used, the table is replaced by a *parameterized function* which can be viewed as a vector $\vec{\theta}$ of parameters. The number of parameters is typically much smaller (often exponentially) than the number of state-action pairs in the original MDP.

The learning setting of VFA is quite different from the supervised, regression setting, where the training data consists of a fixed set of sample points and the task is to fit a function to these training points. Most training methods assume that multiple passes through this dataset are performed to tune the parameters of the function approximation. In the RL setting, the desired outputs for sample points are not available right away, such that there is no static dataset to fit the function to. A second problem is that the data appears to be *non-stationary*, i.e. there is a significant amount of *concept drift* (Maloof, 2003). This is due to the fact the policy generates the learning examples, and this policy can be subject to change, and due to the fact that many methods *bootstrap* such that particular values change constantly. These problems are the reason that VFA is more complex than function approximation in the supervised setting. However, most successful applications of RL use VFA as their main method. Two well-known success stories, Tesauro (1994)'s TD-GAMMON and Crites and Barto (1996)'s *elevator scheduling program* both use *neural networks* for VFA.

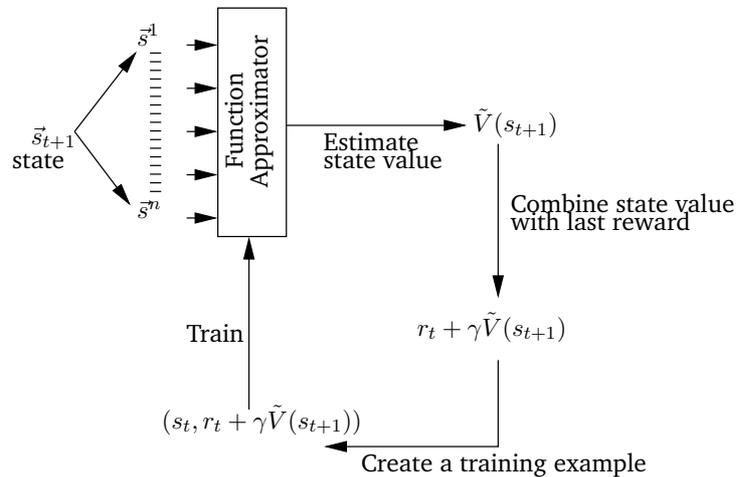


Figure 3.12: The general structure of *value function approximation* (based on state values). A similar mechanism can be used for state-action value functions, in which the input to the function approximator is a state-action pair (s, a) and the output of the approximator is $Q(s, a)$.

Many of the methods for the prediction problem covered in the previous chapter have been described as *backups* in which a particular value was updated in the direction of the backup value. For example, each Q -learning backup, which would backup (or: update) the value of an individual state-action pair, can now be seen as providing a training example for the function approximator. As an example, let us assume that we use the SARSA learning algorithm. When making the transition from state s_t to s_{t+1} by taking action a_t and receiving reward r_t , $Q(s_t, a_t)$ is re-estimated to $Q'(s_t, a_t)$ using the current values of $Q(s_t, a_t)$, r_t and $Q(s_{t+1}, a_{t+1})$, and additionally the discount factor γ . To improve the current estimation of Q , the tuple $\langle s_t, a_t, Q'(s_t, a_t) \rangle$ is used as a training example for the function approximator.

The general structure of using VFA in RL approaches is depicted in Figure 3.12. For the *prediction* problem, for example in estimating a state value function V using TD(0), the current estimation of the value for the state s is updated towards the backup value for s by training the function approximator. For the *control* problem, usually VFA is performed with state-action pairs (s, a) as input, and a state-action value $Q(s, a)$ as output. The update is similar to that for the prediction problem of TD(0). However, for deciding on an action to perform, the function approximator is first used to compute a Q -value for *each* action. Actions are chosen based on these Q -values, following an exploration strategy. Note that VFA over Q -functions *generalizes over actions* as well. A frequently used variation on this approach is to have separate FAs for each action, such that no interference between action values can occur.

The non-stationary nature of the target values in VFA is a complicating factor for *convergence* to a optimal – or at least *stable* – value function. In principle, all algorithms from the Chapter 2 can be combined with VFA. However, the exact *combination* of an approximation architecture with a particular value learning method determines whether convergence can be proved. An important difference is whether *on-policy* or *off-policy* methods are used. In off-policy learning, the target distribution (optimal value function) for the function approximator differs from the distribution from which the samples come from (the value function of the actual policy). For example, Thrun and Schwartz (1993) have shown that errors in function approximation can lead to a systematic overestimation of the Q -function. On-policy algorithms such as SARSA have better convergence guarantees (Gordon, 2000). For many combinations of FAs and RL algorithms *divergence* can occur (see Thrun and Schwartz, 1993; Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998; Gordon, 2000; Ratitch, 2005). Many function approximators have been applied in VFA, such as *neural networks*, *decision trees*, *support vector machines* and *kernels*, and we will discuss several of them in the next sections. Some have better convergence properties (e.g. *averagers* (Gordon, 1995c,a; Szepesvari and Smart, 2004; Wiering, 2004)) than others (e.g. most non-linear methods).

The success of applying VFA depends only partially on the combination of the architecture and the specific RL algorithm. Other factors that are important are the *shape* of the value function and the training method. Most function approximators assume a certain degree of *smoothness* in the value function. *Discontinuous* functions are difficult to learn. For example, consider an extreme case of such a function where the value of a propositional state is equal to the *parity*²⁵ of the bits. The state 0001 will get a value 1 and 0011 a value

²⁵Interestingly, Bakker and van der Voort van der Kleij (2000) show results using recurrent neural networks in these kinds of domains. "Memory" (or some extra representation power) helps in these environments,

0. All pairs of states that differ in only one bit, will have a different value. The shape of this value function is extremely discontinuous and difficult to represent and learn²⁶. (see more on this Thornton (1996); Clark and Thornton (1997); Thornton (2000)).

The *training method* is related to the RL method used, and to whether the problem is episodic or not. In episodic tasks, value updates can be gathered and the function approximator can be updated based on a *batch* of examples (e.g. see Ernst *et al.*, 2005). Batch updates require the *storage* of all examples encountered during this batch, and *retrieval* and *updating* of state-action pairs encountered more than once. This may become a problem when these operations are expensive or when batches are large. Usually, online methods are used though.

Doing RL with VFA requires dealing with the *structural credit assignment* problem, i.e. how to change the adjustable parameters of the approximation architecture with respect to errors in values. A large class of approximation architectures is trained using *gradient descent* techniques. The gradient of an error function is used to move the architecture's parameters such that the error is reduced. Value backups can use partial derivatives to update each architecture parameter. For example, let $w_{i,t}$ be a parameter at time t , and the agent has experienced a transition from state s_t to s_{t+1} while doing action a_t , and receiving reward r_t . The Q -learning update of each parameter $w_{i,t}$ to $w_{i,t+1}$ can be expressed as follows:

$$w_{i,t+1} = w_{i,t} + \alpha \left(r_t + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t) \right) \frac{\partial_t Q_t(s_t, a_t)}{\partial w_{i,t}} \quad (3.4)$$

An extension of this update rule using *eligibility traces* (see Section 2.6.3) attaches an eligibility trace to each parameter. The complexity of the architecture determines the complexity of computing the derivatives. For linear architectures this is rather simple, and for more complex architectures, such as *multi-layer perceptrons* specialized procedures such as *back-propagation* exist to compute weight updates.

The use of VFA is much dependent on the right *bias*, which consists of the FA itself, and the set of *features* it is provided with. Tables are, in that respect, completely *unbiased*, and the update in Equation 3.4 degenerates to the standard Q -learning update (see Equation 2.18) because the weights are the Q -values and the derivative is 1. The use of proper *feature selection* methods and *dimensionality reduction* techniques such as *principal component analysis* (see Reed and Marks II, 1999) can greatly reduce sample complexity, because they usually result in more compact models, i.e. less parameters. An interesting experiment using completely *random* features was devised by Sutton and Whitehead (1993). They showed that a *multi-layer perceptron* using a large number of random input-to-hidden layer weights – which amounts to linear VFA using random features – was able to perform well in RL experiments.

Although the explicit aim of VFA is to approximate the value function (the prediction problem), the implicit aim is to compute an (optimal) policy from the approximated value function (the control problem). In general, a value function does not have to be perfectly accurate in order to derive an optimal policy. This was one of the assumptions in *modified policy iteration* (see Section 2.5.2). The fact that usually *global* error measures such as

although this function can be mapped without using memory.

²⁶However, when making use of the *relational properties* of features, this is actually a very simple problem. Relying on the statistical interaction between features results in a difficult problem. See Thornton (2000) and (Clark and Thornton, 1997) and (Thornton, 1996) on this topic.

MSE are used, makes things more complicated. It is arguable whether finding a minimal MSE is the best way to assess the quality of a value function approximation. The greedy policy derived from an FA with minimal MSE might be outperformed by a greedy policy derived from another FA. Local approximations of value functions can, in this respect, be more balanced in the way they can measure error in the approximation.

3.6.1.1 MODEL APPROXIMATIONS

Value function approximation is the most common way of using function approximation in RL approaches. One can use similar constructions to estimate the transition function and the reward function. In the simple, table-based setting, estimating transition probabilities can be done using *counting* and a *maximum-likelihood model* for all transitions. A similar mechanism works for rewards as well. Monte Carlo techniques (see Section 2.6) estimate these quantities too from direct interaction with the environment. Learning a world model can be useful (e.g. see Drescher, 1991) in more efficient algorithms such as DYNA (Sutton, 1991a), *prioritized sweeping* (Moore and Atkeson, 1993), for model-based Q -learning in dynamic domains (Wiering, 2002), and other methods we have described in Section 2.6.3.

More *compact* prediction models of the next state's feature values (and reward) using the current state, are possible. For example, *2-stage dynamic Bayesian networks* (DBN) (see Jordan, 1999) can compactly *represent* this transition model (Boutilier *et al.*, 2000a). We have seen examples of DBN reward and transition models in *factored models* of MDPs in Section 3.5. Probabilistic models enable *Bayesian conditioning* to update model *parameters*, but usually the *structure* of the DBN, i.e. the probabilistic dependencies of features between two subsequent states, is given. Lin (1992) learned to predict the most likely reward and successor state with *feedforward neural networks*. Großmann (2000) used *constructive neural networks* to *grow* a set of *markers* in a continuous space as a discretized model of the state over which a model was estimated (see also Großmann, 2001).

Other structured models of *action dynamics* can be learned, for example in the form of STRIPS rules (Fikes and Nilsson, 1971), but the literature on learning probabilistic planning rules is relatively sparse. Oates and Cohen (1996) showed results in the propositional setting and in the next chapters we will see results for the *first-order* setting (e.g. Pasula *et al.*, 2004). Related to that is the work by Benson (1996) on learning reactive action models in the context of *deictic* representations. Lanzi (2002) shows the possibility of shifting from a tabular representation of RL problems to a classifier-based representation. In the *learning classifier systems* (LCS) literature (Lanzi *et al.*, 2000), *anticipatory classifier systems* (Stolzmann, 2000) learn structured models to improve their behavior. The YACS classifier system (Gérard *et al.*, 2002) learns compact transition models. Its recent extension MACS (Gérard *et al.*, 2005) too is an anticipatory LCS which combines the model-building and planning capabilities of a DYNA architecture (Sutton, 1991a) with the generalization capability of LCS.

In this section we have discussed approximations of value functions and models. Another approximative dimension concerns *policy approximations*. Policies are mappings from states to actions and these can be represented by an approximation architecture too. One way is to *derive* an abstract policy from an abstract value function, which reduces RL to *classification* (Lagoudakis and Parr, 2003). Other methods *search* for policies *directly* without using explicit representations of value functions. We will discuss methods for *policy search* in Section 3.7.

3.6.2 Architectures for VFA

Many architectures from *supervised classification* and *regression* can be used for VFA. Two prerequisites are that the architecture should support *online* or *incremental* learning, and it should provide means to deal with *concept drift*. Some architectures can be used without major changes, for example *feed-forward neural networks* but some, such as *decision trees* need to be adapted for this specific setting. Because the end result, i.e. the exact function and its complexity, is not known beforehand, one can choose either for a static, general architecture that can handle arbitrary mappings, or for an architecture that adapts its structure on-the-fly based on resource demands during learning. We start with a discussion of commonly used, simple, linear architectures. After that we discuss *non-linear* architectures, both static and adaptive. All architectures assume a propositional, i.e. *feature-based*, state representation, and that a sufficient set of features is selected prior to learning.

3.6.2.1 LINEAR FUNCTION APPROXIMATION

Doing function approximation based on a *linear combination* of features offers a number of advantages. Strong mathematical theory supports their definition, and performing gradient descent techniques over linear approximations can be done very efficiently. These algorithms can converge to a *global optimum*, minimizing the *mean squared error* (MSE) over a static data set. Given a set of N basis functions $\{\varphi_i(s)\}$ defined over state vectors s , and a vector of N parameters θ_i , a linear combination of the value function can be expressed as follows:

$$V(s) = \sum_{i=1}^N \theta_i \varphi_i(s)$$

The choice for particular basis functions is very important for a good approximation.

The simple case where each basis function φ_i projects the state onto the i th feature, computes the output from a weighted combination of feature values, implementing a *perceptron* (Minsky and Papert, 1988) (see also Haykin, 1999). Basic model-free RL approaches based on linear VFA are working on PIAGET-2. However, several approaches have extended linear

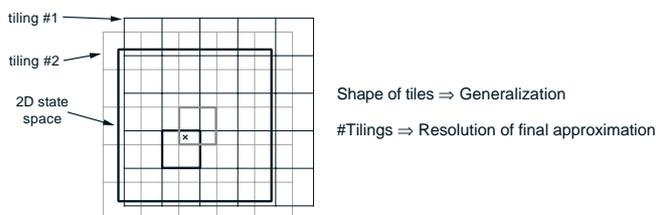


Figure 3.13: CMAC tilings (Adapted from Figure 8.5 in (Sutton and Barto, 1998)).

VFA to the PIAGET-3 case. Miyamoto and Uehara (1999) introduced *feature constructive Q-learning* (FCQL) that uses *gain ratio criteria* as used in decision trees to evaluate and search for new features. Several *constructive function approximation* approaches (Utgoff, 1996; Utgoff and Precup, 1997, 1998; Utgoff and Stracuzzi, 2002) provide means to find new features for VFA approaches. Specifically, the ELF algorithm (Utgoff, 1996) finds new features for a linear approximation by looking at their reward *span* and splitting those features. Extensions along these lines are *multi-layered* approaches (Utgoff and Stracuzzi, 2002) though these consist of multiple layers of features and are more powerful than simple linear architectures.

Model-based (DP) approaches based on linear approximations can be found in the *factored representation* approaches (see Section 3.5). The *approximate value iteration* (AVI)

approach by (Schuurmans and Patrascu, 2001) uses a linear value function approximation in a model-based setting such that good weights for the basis functions can be found by linear programming (where the constraints can be generated using the Bellman equations for example). Wu and Givan (2005) describe an interesting extension to AVI by learning new basis functions (see Parr *et al.* (2007) for related theoretical results). These are learned by sampling states and building features (using a decision tree) for states with either large negative or positive Bellman errors. The weights of the linear approximation are found by Monte Carlo sampling techniques. On a conceptual level this is related to the *cascade-correlation* neural networks discussed in the following sections.

The *cerebellar model articulation controller* (CMAC) (Miller *et al.*, 1990) architecture is a linear, local function architecture often used in RL (Sutton, 1996; Santamaria *et al.*, 1997) and is a form of *coarse coding*. It consists of a number of *overlapping* regions, each of which represents a binary feature. A feature is activated (has a value 1) if the region contains the input state. A popular type of coarse coding is the *tile coding*. It consists of a number of overlapping *tilings* (i.e. partitions), each of which represents a feature (see Figure 3.13). Each *tile*, or grid cell, is activated for some values of a particular feature. Each input vector activates exactly one tile in each tiling. Each tile has a weight associated with it and function approximation is now expressed as a weighted, linear combination of all active tiles. Tile coding can be combined with a hash memory, such that multiple tiles are collapsed into one, but then the model is no longer local.

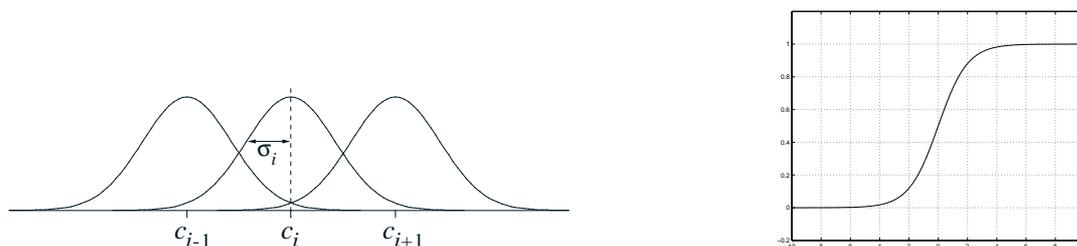


Figure 3.14: a) 3 Radial basis functions (RBF) covering a one-dimensional input space. b) A sigmoid function.

Radial basis function networks (see Haykin, 1999) RBF can be seen as a generalization of coarse codings, to a model containing continuous features instead of binary ones. Regions are now modeled using smooth differentiable functions with a *center* μ and an *activation width* σ . The center is a point in the state space and the width determines the size of the region. The most common RBF is the bell-shaped *Gaussian* function. See for an example Figure 3.14a in which three Gaussian functions model three regions in a one-dimensional input space. Kretchmar and Anderson (1997) compare local approximators such as CMAC and RBFs. Their results show that RBFs have difficulties with the edges of the space and introduce unnatural waveiness as their widths decrease. *Normalized* RBFs displays better narrow width behavior. Additionally they experiment with adaptive local units in the PIAGET-2 sense.

Related to RBF networks are *one-layer sigmoid belief networks* (see Haykin, 1999). These networks have as basis functions *sigmoid* functions, which are smooth, non-decreasing functions within a fixed range, see Figure 3.14b. Each sigmoid is a function defined over the complete state vector. Sigmoid neural networks are essentially non-linear and global, however, for some parameter settings, i.e. if the slope around the symmetry

point is almost vertical, they behave close to binary.

Sparse distributed memories (SDM) (or, *Kanerva codings*, (see Sutton and Barto, 1998, p.209–210)) are a solution to the problem when the number of dimensions of the underlying state space becomes too large. In essence, it separates the complexity of the target approximation from the size and dimensionality of the state space, for example by storing the approximation for a limited amount of *prototype states*. In fact, many of the state space aggregations from Section 3.4 and several local neural network approaches (see further in the next sections) can be seen as implementing some of these ideas. The approach by Ratitch and Precup (2004) (see also Ratitch, 2005, for an extensive discussion on related linear approaches)²⁷ is a recent implementation of these ideas with dynamic allocation and adaption of the SDM resources

3.6.2.2 NEURAL NETWORKS

Neural networks (NN) (Bishop, 1995; Haykin, 1999; Reed and Marks II, 1999) are among the most popular FAs in RL (Lin, 1992; Rummery and Niranjan, 1994; Rummery, 1995; Bertsekas and Tsitsiklis, 1996; Bakker, 2004). NNs, or more generally called *connectionist systems*, are computational structures more or less inspired by the structure and the functioning of the brain²⁸. A great variety of NNs exist, differing in structure, complexity and types of problems they can deal with. NNs are often used for VFA and we treat them in a little more detail.

Two elements that form the core of virtually all NNs are *neurons* and *connections*²⁹. Neurons are the main processing elements of the NN. Usually they are simple computational units, computing an output based on inputs that may come from outside (e.g. problem features), from other neurons, or from recurrent feedback loops. The connections determine the *topological structure* of the network and can be used to structure the *information flow*. Furthermore, one can distinguish between *local representations* in which individual neurons have specific functions and *distributed representations* in which functional structures are distributed over groups of neurons. Another distinction is between standard, *static* neural networks, in which the structure of the network is fixed during learning, and *constructive* approaches, in which the neural structure is expanded and modified during learning.

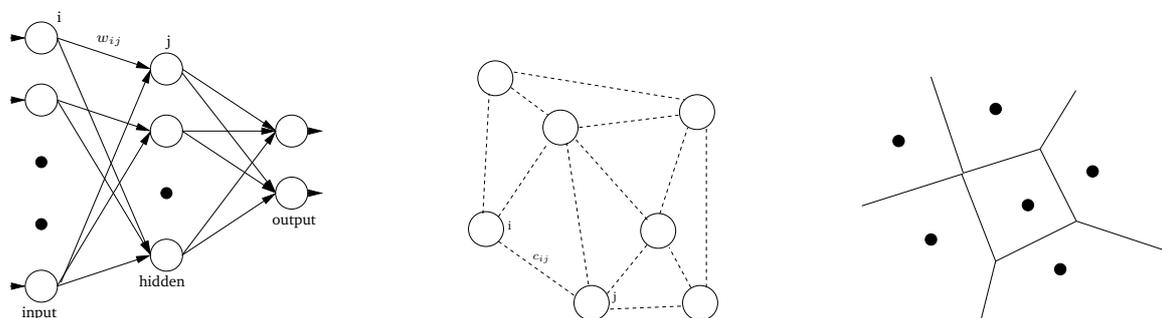


Figure 3.15: a) A multi-layer perceptron. b) A local neural network. c) A Voronoi tessellation.

²⁷Quite interestingly, Ratitch (2005) provides an extensive overview of RL methods rather different from ours. They consider most representations as SDM, including factored representations and decision trees.

²⁸Although many papers in the neural network area claim to be close to and consistent with the computational structures of the human brain, most work only superficially does so.

²⁹Connections in real brains consist of axons and dendrites, but we treat these as one connection.

Static Neural Architectures. In some networks, connections determine the *information flow* through the network. For example, in a large class of networks, the so-called *feed-forward* networks, the information flow is one-directional from an *input layer* to and *output layer*. In Figure 3.15a a *multi-layer perceptron* (see Bishop, 1995) is depicted, which is a *multi-layer* extension of the *sigmoid belief networks*. Each neuron consists of a *sigmoidal function* σ , the so-called *activation function*, that squashes its input smoothly in a range, usually between 0 and 1. There can be *multiple* layers between the input and output layer, although usually there is only one *hidden layer*. Each connection from neuron i to j has a *weight* w_{ij} such that neuron j 's *input* consists of a weighted combination of inputs. The final *activation* a_j of neuron j is computed from applying the activation function to the combination and the neuron's bias³⁰ b_j , i.e. $a_j = \sigma(\sum_i w_{ij}a_i - b)$, where each a_i is the activation of neuron i functioning as an input for neuron j . Feed-forward networks are networks in which only one-directional connections between *subsequent* layers exist. In most practical cases, only connections from one layer to the *next* are used.

Extensions of feed-forward networks can be found by allowing more general connection structures. Networks with connections from one layer to a previous layer, are called *recurrent neural networks* RNN. (Elman, 1990; Tsoi, 1998). In RNNs outputs of some neurons can function as an input for other neurons "earlier" in the network. *Elman* networks contain connections from the hidden layer to the input layer, and *Jordan* networks from the output layer to the input layer. These connections can be delayed an arbitrary number of time steps n , such that the output of the neuron at time step t can function as an input to the network at time step $t + n$. RNNs can model *temporal* structure in the input such that they are particularly useful for POMDPs (see Bakker, 2004). Completely recurrent networks, such as *Hopfield networks* (Haykin, 1999) consist of arbitrary – though usually fully – connected topologies, in which each neuron can function as either input, output or both. Computing an output for these networks is done by *simulation* from an initial *activation* until a stable state, a so-called *attractor state*, is reached. Hopfield networks have no identifiable *layers*, or, in other words, there is only one layer.

Training feedforward networks is usually done using *gradient-based* methods which were explained in the previous section. The *back-propagation* (BP) algorithm (see Bishop, 1995) first computes the *error* between the network's output and the *desired* output as given by the training example. This error is then used to update the weights of the connections feeding into the output layer, using the gradient. After that, the weights of earlier layers are adjusted relative to their partial derivatives. Many efficient extensions of the BP algorithm have been proposed. For example, *resilient* BP (RPROP) uses only the *direction* of the gradient and not the *magnitude* whereas the *Levenberg-Marquard* algorithm uses *second-order partial derivatives* (see Reed and Marks II, 1999). Strictly feed-forward networks can be trained using gradient methods, but *recurrent* networks require more sophisticated methods. *Unfolding* the network, by replicating activations through time, enables the use of algorithms similar to BP to train the weights. However, this generates deep network structures and an additional problem is that the partial derivatives for weights deep in the network diminish, which makes training very slow. The use of more sophisticated gradient methods or *evolutionary techniques* can sidestep this problem.

In a second class of networks, connections are more concerned with the *structure* of the

³⁰For notational convenience, this bias is sometimes modeled as a fixed input -1 such that the weight for this connection functions as the bias.

NN and the representation of the data is *local*. In these networks, connections are treated as *neighborhood structures*, or *distances* between neurons in the input space. Examples are *Kohonen networks*, *radial basis function networks* and *neural gas*, see also Figure 3.15b. Each processing element N_i , sometimes called a *cell*, in such a network is positioned somewhere in the input space, and connections c_{ij} can be present between cells i and j . In so-called *winner-takes-all* (WTA) architectures, the total network output is determined by the cell that is *closest* to the current input (e.g. using an *Euclidean norm*), such that all the cells are in *competition* to deliver this output. As a consequence, the input space is partitioned into separate *regions* each covered by one neuron, e.g. one cell. An example is the *1-nearest neighbor* scheme which gives a *Voronoi* tessellation of the input space (see Figure 3.15c. Other networks based on cell structures are based on *soft cell competition* (SCC). They use *all* cells to compute the network's final output. These methods result in more smooth approximations as the final network output is mainly determined by a weighted combination of outputs of cells that are relatively close to the current input. The k -nearest neighbors algorithm with $k > 1$ is an example of this.

Training WTA architectures is done by *moving* the cells after each presented example. In Kohonen networks, the *winning* cell N_i is moved towards the example input, and its *neighbors* in the neural architecture – determined by the connections, e.g. all cells N_j such that c_{ij} or c_{ji} exists – are moved slightly too. After presenting all examples a sufficient number of times, the NN will have distributed all its cells across the input space such that more cells are present in areas where the density of examples is higher. The same mechanism drives *unsupervised* 1-nearest-neighbor approaches where the final layout of cells is interpreted as representing the *clusters* in the data, where each cell N_i represents the *cluster center*. Training SCC architectures involves learning the weights of the linear combination of cell outputs, and in addition, learning the position and slope of the cell's activation functions.

The distinction between local and distributed representations in neural architectures has some important consequences. For many architectures, such as MLPs, it has been proved that they are *universal approximators* (Scarselli and Tsoi, 1998), i.e. they can represent any mapping $\mathbb{R}^n \rightarrow \mathbb{R}^m$. However, this is about *representation*, and not about *learning* this representation. Learning the weights for an architecture is called the *loading problem*³¹ and it is NP-complete for most cases (see Haykin, 1999; Reed and Marks II, 1999). Choosing for either a local or a distributed representation depends on the function that has to be approximated. If the characteristics of this function vary throughout the input space, local methods often perform better (Lawrence *et al.*, 1996). Another advantage of local methods is that they are less vulnerable to *catastrophic forgetting* (French, 1999). This phenomenon occurs in distributed representations with online learning. In this case, all weights are updated with respect to the error for the current example, such that the output for previous examples is changed as well. As a consequence, the network tends to *forget* old examples, which can only be resolved by frequent retraining. Related to this is the *herd effect* (Fahlman and Lebiere, 1990). Because all weights are changed with respect

³¹The *loading problem* has the following specification. *input*: a network architecture and a training set. *output*: determination of the network weights such that every output in the training set is mapped onto its desired output, or a message that this is not possible. The loading problem is NP-complete for the *classification* setting. For the regression setting a similar result is not known, but for finite training sets it is equivalent to the classification setting.

to the same current output error, it can take a long time before neurons start to *specialize*. A difficulty with all neural architectures is how to choose the right number of *neurons*, i.e. the *architectural bias*. Small networks might not be capable to learn a sufficient approximation of the target function, whereas large networks are vulnerable to *overfitting*. However, recent findings for MLPs suggest that when simple gradient-based approaches (e.g. standard BP) are used, online learning can result in a smooth approximation, even for very large networks (Caruana *et al.*, 2001). Other techniques to prevent overfitting are *regularization* in which an additional penalty term (e.g. with respect to weight values) encourages smoother approximations, and *early stopping training* methods in which a separate *validation* dataset is used to monitor the overfitting such that training can be stopped when the network starts to overfit the data (see further Reed and Marks II, 1999).

Knowledge-Based and Hybrid Neural Structures. Most (distributed) NNs belong to the *sub-symbolic* paradigm. The sub-symbolic representation level consists of the combined *activations* of neurons, where the *individual* neurons have no immediate *symbolic* interpretation (see Smolensky, 1989). The hidden layer(s) of the network build up a new representation of the input, i.e. a new set of features. This in contrast to *symbolic* representations such as propositional *rules*, where every symbol has a clear *semantics* with respect to the problem domain. A number of approaches has studied the *incorporation* and *extraction* of symbolic knowledge in the context of (global) NNs (Sun, 1998; Cloete and Zurada, 2000). One of the most well-known methods is the *knowledge-based artificial neural network* (KBANN) approach (Towell and Shavlik, 1994). It uses an *a priori*, reasonably correct, *propositional knowledge base* which is translated³² into a NN structure with weights. This NN is then trained through standard BP to fine-tune the weights. From the trained network, an improved rule-based system can be extracted. Many other methods have been proposed to *extract* symbolic knowledge from trained NNs (see d'Avila Garcez *et al.*, 2001).

Hybrid neural structures combine NNs with other representational formalisms, or use NNs as substructures in larger systems. The CLARION (Sun and Peterson, 1998) architecture is a 2-level architecture that combines neural *Q*-learning to learn *procedural* knowledge and a set of propositional rules that represent *declarative* knowledge extracted from the neural network. *Adaptive Neuro-Fuzzy Inference Systems* (ANFIS) (Jang, 1993; Jang *et al.*, 1997) are similar to the KBANN method, although the rule base now consists of *fuzzy rules*³³ instead of *crisp* propositional rules. The *Evolving COnnexionist Systems* (ECOS) by Kasabov (1998) and the *Dynamical Recurrent Associative Memory Architecture*

³²A propositional rule base can be encoded into a feed-forward neural network. Sigmoid functions are – in the limit – simple *threshold* functions. *Conjunctions* and *disjunctions* of propositions (e.g. features) can be implemented by an appropriate choice of weights. For example, the function $a \vee b$ can be implemented by ensuring that the weights w_a and w_b for inputs a and b are both larger than the *bias* of the neuron. In this way, activation of either a or b will activate the neuron, thus implementing the OR of a and b .

³³*Fuzzy Logic* (see Jang *et al.*, 1997, for pointers to the literature) is a popular paradigm for dealing with *subjective probabilities* and *vague*, or *fuzzy* concepts. Let us consider the fuzzy concept "old". It can be represented by a continuous function between 0 and 1 over a domain of *ages*, for example ranging from 0 to 100. The exact slope of this function is dependent on a subjective measure of "how old" each age is considered. An age of 10 might be considered almost zero on a scale of 0 to 1, an age of 60 might be considered 0.7 and an age 100 is close to 1. Fuzzy logic approaches are able to *reason* with these fuzzy variables. For example, one can infer a fuzzy value 0.8 for *grey* based on fuzzy values of 0.9 and 0.67 for fuzzy variables *old* and *bold*. Fuzzy logic is very popular in control theory and many modern micro waves or brake systems for cars contain fuzzy logic implementations. In fact, the author's washing machine has a

(DRAMA) by Billard and Hayes (1999) are examples of larger systems that incorporate connectionist structures, for example for robotic control and speech recognition. More examples of *hybrid neural systems* have been proposed in the literature (see e.g. Wermter and Sun, 2000). Hybrid combinations of NNs with first-order logic representations will be described in the next chapter.

Constructive Neural Architectures. One of the problems of static neural architectures is that the *topology* has to be chosen a priori. A more fruitful way would be to *learn* this structure as well, resorting in *neural constructivism* (Quartz and Sejnowski, 1997; Quartz, 1999). The last two decades many *constructive* approaches for neural networks have appeared (Kwok and Yeung, 1997; Quinlan, 1998; Parekh *et al.*, 2000; Westermann, 2000; Shultz, 2003). Constructive³⁴ approaches start with an initial topology that gets modified during learning. The general idea of *constructivist* learning is that the *hypothesis space* grows and enables the learner to construct more complex hypotheses on the basis of simpler ones, corresponding to the developmental theory by Piaget (1950) and relating to other work on *constructive function approximation* (Utgoff and Precup, 1998; Utgoff and Straczuzi, 2002). There are several advantages of constructive approaches (Kwok and Yeung, 1997). First, it is easy to specify the *initial network*. Second, constructive approaches search for smaller networks first. This has the added benefits that less data is needed to train the networks for good generalization and that often the algorithm will find smaller networks.

Basically there are three approaches to neural constructivism. *Unsupervised* constructivist approaches (Fritzke, 1997) that grow local, unsupervised neural architectures and *supervised* approaches (Fiesler and Cios, 1997) that grow mainly *feed-forward* NNs such as MLPs, are two classes of *ontogenetic networks*. A third class consists of the use of *evolutionary algorithms (genetic)* (Porto, 1997) to construct NNs, mainly for feed-forward and recurrent NNs.

Constructivist algorithms for feed-forward networks work by adding nodes and weights. The *Dynamic Node Creation* (DNC) algorithm by Ash (1989) adds hidden neurons during learning each time the network's error asymptotes and the error is still unacceptably high. The *meiosis networks* approach by Hanson (1990) *splits* nodes with high weight variances and the *node splitting* algorithm by Wynne-Jones (1991) splits nodes based on *oscillating* weights. The well-known *cascade correlation* algorithm by Fahlman and Lebiere (1990) (see also Prechelt, 1997) uses a more complex procedure for adding nodes. The initial network consists of an input and output layer, fully interconnected. In the *output phase*, all the weights leading to the output units are trained, until no improvement is possible. In the *input phase*, a *pool* of hidden units, only connected to input units and previously inserted hidden units, is trained to maximize covariance of their output with the remaining network error. The one unit with the highest correlation is inserted in the network as a new hidden unit, and its input weights are *frozen*. This unit is fully connected with the output units. Training of the network then continues with another *output phase*. This procedure is repeated until network performance is at an acceptable level. Benchmark

label attached to it, ensuring it uses fuzzy logic (for determining the amount of water, given the weight of the current content).

³⁴We use the term *constructive* in a broader sense though. *Pruning* algorithms, that begin with a large structure and decrease its size during learning, do essentially the opposite of constructive approaches. Nevertheless, we will consider pruning as a form of constructive learning.

tests with CC showed that it outperformed BP on several tasks (Fahlman and Lebiere, 1990). More recent extensions of these ideas are *constructive back-propagation* (Lehtokangas, 1999), which supports more efficient computational algorithms and the addition of *multiple* hidden units per input phase, and *knowledge-based cascade correlation* (Shultz and Rivest, 2001; Shultz, 2003), that allows for the insertion of previously learned networks. Prechelt (1997) investigates some of the earlier CC approaches. All these approaches *extend* the neural architecture by adding nodes, differing in the criteria for doing that and whether parts of the network get frozen. An example of a constructivist version of fuzzy NNs is the EFUNN architecture by Kasabov (2001). Many more constructive approaches for feed-forward and recurrent architectures have been proposed (see Kwok and Yeung, 1997; Fiesler and Cios, 1997; Reed and Marks II, 1999; Westermann, 2000, for pointers to the literature).

Many constructive approaches have appeared in the *unsupervised* setting. The unsupervised, local setting has some advantages for constructive methods. Because neurons operate more *locally*, adding new neurons usually does not affect approximations in other regions of the input space. Structural adaptations can include the addition of neurons and the modification of neighborhood structures. The *Growing And Learning* (GAL) network (Alpaydin, 1991) implements what is basically a nearest neighbor algorithm for the classification of input patterns: when an input does not belong to the same class as the inputs covered by its nearest hidden unit, a new unit is inserted for this input. More sophisticated methods that cover an input space with local receptive fields, i.e. using *cell structures*, are *growing cell structures* (GCS) (Fritzke, 1994), *growing neural gas* (GNG) (Fritzke, 1995), *dynamic cell structures* (DCS) (Bruske *et al.*, 1997) and *life-long learning cell structures* (LLCS) (Hamker, 2001) (see (Heinke and Hamker, 1998; Hamker, 2001) for some comparisons). These *cell structures* have as main advantage that no a priori decision is needed about the network size. They *cluster* the input space like RBF networks, but the nodes are organized within a graph in which the centers and the connecting edges are updated online. As an example, in GNG new units are inserted between the unit with the highest local error and its highest-error topological neighbor. The idea here is that a unit which is responsible for a large area of the data space will accumulate a high error because it is the nearest unit for far away inputs, and will therefore become the preferential location for the insertion of new units. Similarly, a unit in a very dense area of the data space will accumulate error through a high number of short distances to the data items for which it is the winner, and more units will thus be inserted in such a region as well. With these cell structures, the network architecture comes to faithfully represent the distribution and density of the data items. GNG, DCS and LLCS continuously adapt the *neighborhood structure* during learning, whereas GCS is restricted to a fixed topology dimension. All methods can be equipped with different learning rules and activation functions. LLCS dynamically adapts the learning rate at each node separately. The general challenges in all these constructive architectures are useful *stopping criteria* for node insertion and criteria for moving neurons and adapting neighborhood structures. Many other methods for growing networks of local units have been proposed (see Fritzke, 1997; Westermann, 2000; Hamker, 2001, for overviews).

Reinforcement Learning using Static Neural Architectures. Many successful applications use static NN architectures for RL problems. For example, MLPs are a popular choice for VFA in the model-free setting because they are supported by many software systems,

are capable of incremental, online learning, and can be naturally incorporated in many RL algorithms. Successful applications that use NNs are Tesauro (1994)'s TD-GAMMON and Crites and Barto (1996) elevator scheduling problem but there are many more approaches in areas such as fingerprint recognition (Bazen *et al.*, 2001, and see also Section 3.9), robot control (Thrun, 1996) and game playing (see more on this Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998). A general distinction between methods is that feedforward or recurrent networks are often used for VFA and policy gradient methods (see Section 3.7) whereas local architectures are often used for state space discretization and transition models.

The approaches in this section work at the PIAGET-2 level, in which an architectural bias is given, but the structural credit assignment problem involves the adjustment of parameters of the NNs during learning. In the classification setting, an important problem is *overfitting*. In the RL setting, overfitting is usually not considered a large problem, mainly due to the fact that learning is often online and simple gradient methods are used for updating parameters, which result in incremental, smooth approximations (see Caruana *et al.*, 2001). However, because the data in RL is more or less non-stationary, the main problem is the *convergence* to a stable, or optimal, value function. Early work by Thrun and Schwartz (1993) showed, both experimentally and theoretically, that a *systematic overestimation of values* is a problem for FAs such as NNs, and the main reason for this is bootstrapping (see Sutton and Barto, 1998). Boyan and Moore (1995) designed a more robust algorithm based on *policy rollouts* instead of bootstrapping. An additional problem with NNs for VFA is *catastrophic forgetting* (French, 1999). Especially when off-policy methods are used, the distribution with which the network is learned, is different from the target function³⁵. Despite theoretical problems involving convergence, NNs have been shown very useful in practice for VFA.

Neural networks, mainly MLPs, have been used for Q -learning (Lin, 1992, e.g), SARSA (Rummery and Niranjan, 1994; Rummery, 1995) and many other variants such as $Q(\lambda)$, *adaptive heuristic critic* and SARSA(λ) (see more on this Bertsekas and Tsitsiklis, 1996). The work by Abul *et al.* (2000) describes interesting experiments comparing various combinations of MLPs, eligibility traces and algorithms such as Q -learning and SARSA in a multi-agent setting. The algorithms are compared by letting multiple agents interact in a grid world where the agents have to compete for food while avoiding obstacles. This environment is similar to the one introduced by Lin (1992). Sutton and Whitehead (1993) showed that the difficult task of learning the weights for MLPs can be simplified by using *random* input-to-hidden weights, thereby simplifying VFA to a linear approximation problem. The *explanation-based* NN approach by Thrun (1996) and the *multi-task* learning approach by Caruana (1997) both show how learning an MLP's parameters can be based on multiple tasks. For example, multiple tasks can share the same hidden layer of an MLP such that intermediate features useful for multiple tasks are learned (Caruana, 1997). Finally, Anderson (1993) introduces *restarts of hidden neurons*. A network containing one hidden layer of neurons with Gaussian activation functions is used to approximate a Q -value function. Based on estimated errors and coverage of inputs, the least useful hidden neuron is then *restarted*, i.e. its weights are set to new values (which causes the neuron to be *relocated* in the input space).

Recurrent NNs can learn *temporal* aspects of a task such that they are often used in

³⁵On the other hand, forgetting is can be useful for VFA because the target values constantly change.

tasks with temporal structure in perception-motor interaction loops (Ziemke, 2000) and in POMDPs (see for a good overview Bakker, 2004). The presence of temporal dependencies in the perceptual input stream of the agent necessitates dealing with the *road sign problem*, i.e. for a robot navigating in an environment it may become necessary to *remember* a perceptual input from the past (e.g. a road sign) to make a good decision on the next action. The use of NNs in this context has been described extensively by Rylatt and Czarnecki (2000). Bakker and van der Voort van der Kleij (2000) show that even when there are no temporal patterns in the input, recurrent connections might still be useful in learning. The use of *hybrid* neural networks in RL was considered in the CLARION system (Sun and Peterson, 1998; Sun, 1998) in which an MLP learns *procedural* knowledge using online RL, and a *propositional rule* layer extracts *declarative* knowledge from the MLP. Similar in spirit is the *Fynesse* architecture by Riedmiller *et al.* (1999) that uses a *fuzzy* representation of a controller (i.e. a policy) combined with a NN performing VFA.

Reinforcement Learning with Constructive Neural Architectures. Although static NNs for RL are of high practical value, supplying the architectural bias can be difficult. Especially in the RL setting, the target data is not available from the start, such that the optimal *size* and *topology* of the network are not yet known. Usually this is solved by using large network structures. However, constructive approaches can – in principle – grow during learning, adapting to the task’s needs. A challenge in the RL setting is to determine when to stop growing. This is the *stability-plasticity dilemma*³⁶ and it denotes the trade-off between a keeping a flexible structure (*plasticity*) and fixating the structure to reduce variance (*stability*). Some algorithms *freeze* weights (decreasing plasticity), which may not be most suitable in the RL setting, where target values are not fixed throughout learning. Constructive NN algorithms belong to the PIAGET-3 level, learning both the structure and the parameters of the architectures. The use of constructive algorithms for global NNs in VFA is limited so far. One of the reasons is the non-stationarity of RL data which creates problems for incremental growing of global architectures. As described, most algorithms apply some form of weight freezing, which *fixes* parts of the mapping, whereas in RL it is necessary to keep a sufficient amount of *plasticity*.

Ring (Ring, 1994, 1997) developed a constructive, *higher-order* neural network, known as *temporal transition hierarchies* (TTH), that has been used as representational tool in conjunction with (multi-task) RL. There are two types of units in TTHs. *Primitive units* are the input and output neurons of the network. *High-level units* enable the network to change its behavior dynamically, by modifying strengths of connections. TTHs learn context-dependent tasks where previous inputs affect future actions. They do this by adding new units to examine the temporal context of its actions for clues that help predict their correct *Q*-values. When new units are added to the network, they are built on top of the existing hierarchical network structure to modify existing transitions. When no high-level units are present, the transition hierarchy network behaves like a simple, single-layer feed-forward neural network. New high-level units are created during learning. If one unit is reliably activated after another, there is no reason to interfere with the connection between them. Only when the transition becomes unreliable is a new unit required. This is the case when the connection weight should be different in different circumstances. A new unit is added whenever a weight is forced to increase and decrease

³⁶This is very much related to the exploration-exploitation dilemma in the RL algorithms of Chapter 2.

in the same temporal context. The additional unit is created to determine the contexts in which the original weight is pulled in each direction. TTHs are linear, but given previous outputs, the network can compute non-linear outputs. Furthermore, they are capable of learning k -Markov tasks, (with unknown k). The work on TTHs also appears in the work by Großmann and Poli (see Großmann and Poli, 1997; Großmann, 2001).

Two other global, constructive NN in the context of VFA are based on the CC algorithm (Fahlman and Lebiere, 1990), which was designed for the supervised learning setting. Rivest and Precup (Rivest and Precup, 2003; Bellemare *et al.*, 2004) extend the CC to the online RL case by introducing a two-stage process. In the first stage, the agent selects actions and gathers inputs with accompanying TD values in a *cache*. Once the cache is full, the network is trained using standard CC. After training, the cache is emptied and learning returns to the first stage. Experiments in e.g. TIC-TAC-TOE show the potential of this *semi-online* approach. A recent, *fully online* application of CC in RL was proposed by Vamplew and Ollington (2005b). This method uses separate networks for each action, and can be applied in many algorithms such as Q -learning and SARSA, using eligibility traces. Multiple candidates for new neurons are trained *in parallel* with the network that is being used, as if they were part of the network already (although they are not used for the network's output). A *patience* period is awaited before adding the best candidate neuron to the network. The performance of the approach is less stable than local, constructive methods, though it gives more compact solutions.

Local, constructive architectures are more amenable to the incremental, online RL setting. Usually they are based on local cell structures (see also Bruske *et al.*, 1997). Großmann (Großmann, 2000, 2001) uses *constructive cell structures* (CCS), based on *growing cell structures* (Fritzke, 1994), to learn a state space discretization. The criterion of GCS was modified for use in RL contexts. The initial network is obtained by random exploration in the environment. Inserting nodes is done by creating a *fringe* node on a selected *parent* node, and using unsupervised learning to place both neurons in the input space. Neurons can also be removed from the topology. Over the current input space discretization (i.e. the cells) a model is estimated and value for all cells are computed. Splitting a parent node is then done similar in spirit as the UTREE approach (see next section). Bruske *et al.* (1997) describes an approach that is amenable to a fuzzy interpretation of the controller. It uses *dynamic cell structures* (DCS) and REINFORCE (Williams, 1992) (see also Section 3.7) gradient-based learning for adapting the network topology and parameters. Experiments in a robotic *obstacle avoidance* task showed stable results in spite of a continuing plasticity of the network. Vamplew and Ollington (2005a) compare *local* and *global* constructive FAs for use in RL and concludes that global FAs may be more unstable, but their final result is usually more compact. As a local method, they use an adaptation of the *resource allocation networks* (RAN) (similar to Anderson, 1993).

Reinforcement Learning using Neuro-Evolution. Although NNs have been proved successful in VFA, and constructive approaches alleviate some of the problems of providing the right architectural bias, still learning both the structure and parameters of NNs using online learning is a hard problem, and usually very slow. *Evolutionary approaches* (Holland, 1975, 1986; Goldberg, 1989; Booker *et al.*, 1989; Bäck, 1996; Mitchell, 1996) have been used for the *evolution* of MLPs (see Porto, 1997; Yao, 1999; Reed and Marks II, 1999; Parekh *et al.*, 2000, for overviews). An evolutionary algorithm searches for good networks directly in the space of all NNs (see also Section 3.7). This space is usually restricted to

networks of a certain size. In each iteration of an evolutionary algorithm, a set of network structures (the *population*) is kept. For each network (*individual*) a score (*fitness*) is computed based on its performance on the current task, for example classification accuracy on a dataset. All high-valued individuals are then used to create new individuals, by combining partial structures of two individuals (*recombination*) or by modifying some parts of an individual (*mutation*). Recombination³⁷ for NNs can be performed by swapping parts of the network, and permutation can be done by modifying weights and biases. The new population is presumed to contain better individuals. This procedure is repeated until some of the population's individuals reach an acceptable performance (see more details on evolutionary algorithms in Chapter 5).

Evolution of NNs has been used in the RL setting (Moriarty *et al.*, 1999). The *fitness* of individuals can be translated in terms of rewards obtained in the target domain. Advantages of evolutionary techniques are that they do not use gradients, that they are applicable to *non-Markovian* domains (i.e. partial observability is not a problem) and do not bootstrap. *Memory* (to deal with partial observability) is easily incorporated by allowing the evolutionary algorithm to work on *recurrent* connections too. Possible disadvantages are that it is difficult to assess, and domain-dependent, how many *trials* are needed to reliably compute the fitness. Because the environment (and rewards) may be stochastic, the fitness may vary much.

A successful neuro-evolution approach is SANE (Moriarty and Miikkulainen, 1996, 1998). It keeps two levels of structures, that of neurons and that of *blueprints* of networks. Evolution takes place on both levels, and blueprints represent good combinations of neurons. SANE has been shown to be very efficient for learning sequential decision making. A crucial assumption is that the network structure has a fixed size and topology and the algorithm learns *weights* within structures. An interactive, online version was described by Agogino *et al.* (2000) and impressive results for the game GO were obtained (Richards *et al.*, 1998). Another interesting application of SANE was performed by Moriarty *et al.* (1998) who evolved agents representing car drivers on a high road, where the aim was to optimize traffic flow. Agents were supplied with a *desired speed* and individual policies were evolved for every driver³⁸.

SANE evolves the weight structures of fixed topology networks. The approach was extended (Kaikhah and Garlick, 2000) by allowing the *size* of the networks to be optimized too. A recent extension is NEAT (Stanley and Miikkulainen, 2001), that allows for the evolution of *topologies* of NNs. An important question is whether evolving topologies along with weights provides an advantage over evolving weights on a fixed topology. Because, a fully connected network can in principle approximate any continuous function (Scarselli and Tsoi, 1998). Stanley and Miikkulainen show that if the problems of how to *represent* variable-sized topologies and how to *minimize* the topologies *during evolution* are solved, neuro-evolution over weights and topologies can decrease learning time considerably. SANE can be seen as acting at PIAGET-2 whereas NEAT at PIAGET-3.

³⁷In most practical systems, the recombination of partial networks is not used, because swapping parts between two good networks often results in a much worse, new individual. This is mostly due to the *distributed representation* in MLPs and related networks.

³⁸Interestingly, they showed that agent that were supplied with a high desired speed would drive on the left side of the road (there were three lanes) and slow drivers on the right.

3.6.2.3 DECISION TREES

Decision trees (DT) (Breiman *et al.*, 1984) partition the input space into variable size rectangular regions. Analogous to the general FA setting, *classification trees* are used for target functions with a finite, discrete set of possible values, whereas *regression trees* are used for real-valued targets. An example tree and corresponding input space partitioning is depicted in Figure 3.16. The *internal nodes* of the tree contain *tests* whereas the *leaf nodes* contain the function's output values. *Binary trees* use only *boolean tests*, such that each node contains exactly two *child nodes*. In order to find the predicted value for an input vector \vec{x} , one traverses the tree, starting from the *root node*, following the outcomes of the tests. The tree's output is then found as the value of the leaf node reached. Supervised training of DTs builds up the tree in an incremental fashion. The root node is constructed by finding a test that splits the dataset in two parts, thereby maximizing *information gain*, typically using some form of *entropy* measure that characterizes the *impurity* of the two parts (see Mitchell, 1997, Ch.3). Then, iteratively the leaf nodes of the tree are split until the tree's predictive accuracy reaches an acceptable performance. Note that decision tree learning algorithms do not *backtrack* such that choices for tests are never reconsidered (but see Utgoff, 1997, for a *tree restructuring algorithm*).

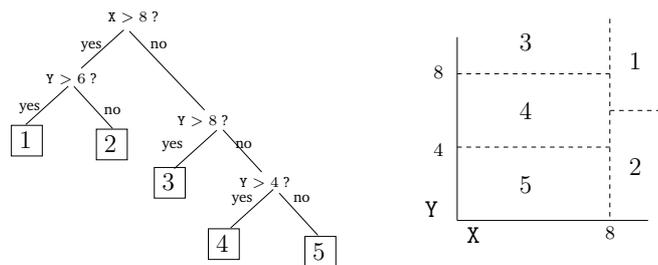


Figure 3.16: a) A *Decision tree*. b) A *state partitioning* induced by the decision tree in a 2-dimensional input space.

Decision tree algorithms have been used for VFA in RL settings. The main difficulty with DTs in this setting is that choices for tests are final, which can be problematic given the non-stationarity of the data used to build the tree. The *splitting criterium* for building the trees has to be adapted to the way examples are generated in an RL interaction. The *G*-algorithm by Chapman and Kaelbling (1991) is an *incremental regression tree algorithm* for *binary* feature vectors that tackles this problem by using a *statistical test* on the *relevance* of individual features for predicting *Q*-values. It uses the standard *Student's t test* which can be used to estimate the probability that after the split, the two resulting partitions have distinct distributions. A crucial assumption is that features are relevant *in isolation*, such that performance is severely affected by problems in which *groups* of features are collectively relevant. This approach was extended by Pyeatt and Howe (1998) who removed the assumption of binary input features, and additionally tested a number of statistical tests for node splitting. The UTREE algorithm (McCallum, 1995; McCallum, 1996) is an incremental tree builder that uses an estimated model to perform value updates for leaves, and to make more informed decisions for splitting leaf nodes. It was later extended to continuous domains (Uther and Veloso, 1998). Finally, Wang and Dietterich (1999) study efficient induction of regression trees using a splitting criterion that is based on differences between values of states that are separated by one step. Additionally, linear functions are

used in the leaf nodes.

Note that using DT for VFA always includes structural adaption of the architecture. Statistical tests enable incremental learning, although one might also consider 2-phase learning algorithms in which RL-phases are interleaved with tree-building phases based on the examples gathered in the previous RL-phase. This method was used in the logical setting by Džeroski *et al.* (1998). Ernst *et al.* (2005) describe tree learning using *batch* methods, thereby reformulating the RL problem into a sequence of supervised problems. DTs are most useful for VFA if the target values are only influenced by a subset of the features and if their influences are orthogonal. Some comparisons between DT methods such as the *G*-algorithm and UTREE were described by Finney *et al.* (2002b) (but see also Finney *et al.*, 2002a).

3.6.2.4 OTHER ARCHITECTURES AND ISSUES IN VFA

In the preceding sections we have described architectures that are used often. In principle, all architectures used for regression in the supervised setting can be used for RL. For example, *kernels* and *Gaussian processes* have been used for VFA by Ormoneit and Sen (2002) and Engel *et al.* (2005). *Support vector machines* were used for batch RL by Dietterich and Wang (2002) and (constructive) architectures of propositional rules with linear VFA by Utgoff (1996). The recent work by Mahadevan (2005a) takes a principled approach called *proto-RL*. Instead of learning from rewards, so-called *proto-value functions* are learned from analyzing the topology of the state space. A mathematical framework supports the formation of *task-independent* basis functions as building blocks of all value functions on a given state space manifold. Other approaches use *instance-based* (e.g. Smart and Kaelbling, 2000) or *memory-based* algorithms (e.g. Moore *et al.*, 1995).

In this book we are mostly concerned with *representational* issues, and likewise when we deal with VFA. Important dimensions distinguish between local and global approximators, batch or online construction of the value function, and properties and complexity of the chosen VFA scheme. When dealing with the *algorithmic* side of algorithms, some of the main issues are whether on-policy or off-policy algorithms are used, whether bootstrapping is employed and whether (or how) convergence can be ensured. Abstraction additionally influences the *convergence* of algorithms. Convergence in MDP contexts means that an algorithm for computing value functions and policies is guaranteed to achieve stability in the learned structure. Often one loses convergence guarantees when using abstraction. In the scope of this book it is not possible to provide a full description of these issues. Some convergence results are known for specific combinations of function approximators and solution techniques for MDPs (Bertsekas and Tsitsiklis, 1996; Papavassiliou and Russell, 1999). Other restricted classes of approximation techniques, under the name of *aggregation* and *averaging* (Gordon, 1995c; Wiering, 2004) allow for stronger convergence results in the face of abstraction (Singh *et al.*, 1995; Li *et al.*, 2006). Additionally, convergence when using abstraction does not have to mean converging to an *optimal* solution. Policies are optimal *on the level of abstraction*; it depends on the quality of this level whether the policy is also optimal on the level of individual states.

For many of the architectures that can be used for VFA no theoretical guarantees can be given for their convergence when used with a particular RL algorithm. Ratitch (2005, Sections 2.4 and 2.5) gives a summary of many results that are known about convergence of algorithms and their complexity. There are several counterexamples for very simple

problems in which the value function approximation does not converge, or even diverges (e.g. see Thrun and Schwartz, 1993; Baird, 1995; Boyan and Moore, 1995; Bertsekas and Tsitsiklis, 1996; Tsitsiklis and van Roy, 1997; Sutton and Barto, 1998). Convergence analysis of RL and DP methods has always been an important and active research direction. However, in many recent directions that use abstractions to cope with huge problems, many approaches aim at *approximations* and solutions that are "reasonably" good, which is called *satisficing*. For example, several hierarchical RL methods focus on *recursively* optimal solutions which may be sub-optimal.

A key assumption in the derivation of error bounds on value functions is that the distribution of training examples is the *on-policy* distribution, i.e. training examples are generated by following the policy being evaluated with. This means that a different formal approach will be necessary to derive bounds for function approximation methods with an off-policy distribution such as Q -learning or DP. In general we would like to follow *one* policy, but preferably learn about *many* policies in parallel, and for this we need off-policy algorithms. Off-policy algorithms are essential for many types of learning algorithms, for example when using temporal abstraction, modularity and macro-actions in hierarchical RL (see Section 3.8).

3.7. ABSTRACTION TYPE IV: Searching in Policy Space

In addition to learning models and value functions (see Section 3.6), policy approximations can be learned too³⁹. Policies are mappings from states to discrete *actions* such that real-valued function approximation is replaced here by either *classification mappings* or *probability distributions* over actions. There are several problems with value function approximations for large domains. They can be difficult to represent, convergence is often not guaranteed, and they are typically more complex than policies. Policy representations usually can be more compact than value functions. Policies can generalize over states that have different values, but the same optimal action. For these reasons, it can be useful to focus on learning policies *directly*, without explicitly representing value functions. Consider a task where the only state variable f can take on values 0..10 with 10 being the goal state, and the actions are `left` and `right` that increase and decrease the f -value. In a discounted setting, the optimal value function for all non-goal states contains 10×2 values, whereas the optimal policy can be compactly represented as *if not in the goal state, do right*. Anderson (2000) shows experimentally that sometimes learning a policy is easier than learning a value function (and deriving a policy from this) in the same domain. Furthermore, policy search is more generally applicable in contexts where only partial information about the state is available, for example in *non-Markovian domains* (e.g. POMDPs), where the Bellman equation does not apply. An additional reason for policy search is that usually there is no need to visit all parts of the state space for obtaining a good policy, as opposed to value-based methods.

There are several ways to search for policies without explicit representations of value functions. On PIAGET-2 level, one can estimate parameters for a fixed policy structure. This is the underlying idea in *policy gradient* approaches. On PIAGET-3 level, there are two ways. One can gather $\langle s, a \rangle$ examples that can be used to *induce* a policy structure

³⁹Humans search in program space whereas as planner searches in problem space. People solve whole classes of problems whereas such a planner uses search to solve only one instance (Baum, 2004).

by supervised learning. Another way is to *search* the space of policy structures, using *evolutionary algorithms* for example. The core problem for all policy search methods is how to estimate *policy quality* to guide the search. A policy can generalize over multiple states having different values, and due to the stochasticity in the environment, estimates of policy quality can vary much.

RL as Supervised Learning. Policy search techniques focus on the policy *improvement* step, where the *evaluation* step is computed without the need for an explicit *representation* of the value function. In order to get an estimate of the policy quality (i.e. the implicit value function) other means can be used. Once good examples of (optimal) state-action pairs can be obtained, standard *classification* techniques can be used to induce a policy structure. In this way, RL can be seen as a (sequence of) classification problem (see Barto and Dietterich, 2004; Langford and Zadrozny, 2005, and Section 3.6.2.3). Lagoudakis and Parr (2003) use an *approximate policy iteration* (API) framework, in which *policy rollout* (Boyan and Moore, 1995) is used to *simulate* the current policy in order to get estimates for a small set of representative state-action pairs. From the obtained examples one can compute $\langle s, a \rangle$ pairs that are used to train a *classifier* representing the policy. Fern *et al.* (2003) use a similar mechanism for *relational* problems and Khardon (1999a) too used inductive policy selection in relational domains (see more on this in the next chapters).

Policy Gradient. A principled approach to policy search are so-called *policy gradient* (PG) techniques, which use gradient-descent on the policy structure itself. The basic assumption is that the policy is represented by an architecture for which a *gradient* (e.g. w.r.t. a Q -function) can be determined for its parameters, for example an MLP. The problem with deterministic policies however, is that these mappings are *discontinuous* for discrete actions. Usually one takes a stochastic policy, for example the *softmax function*⁴⁰ (see also Chapter 2):

$$\pi_{\theta}(s, a) = \frac{e^{Q_{\theta}(s, a)}}{\sum_{a'} e^{Q_{\theta}(a', s)}}$$

For deterministic environments, one can improve the policy along the direction of the PG vector $\nabla_{\theta}\rho(\theta)$, or by hill-climbing, i.e. following an *empirical gradient* direction. For stochastic environments, the *variance* between trials can be large, which makes following the gradient more unreliable. However, one can obtain an *unbiased* estimate of the PG at θ , $\nabla_{\theta}\rho(\theta)$ directly from the trials executed at θ . The true gradient of the policy value based on N trials can be computed by (Williams, 1988, 1992):

$$\nabla_{\theta}\rho(\theta) \approx \frac{1}{N} \sum_{j=1}^N \frac{(\nabla_{\theta}\pi_{\theta}(s, a_j))R_j(s)}{\pi_{\theta}(s, a_j)}$$

for each state s visited, where a_j is executed in s on the j th trial and $R_j(s)$ is the total reward received from state s onwards in the j th trial. This is the REINFORCE algorithm, and it forms the inspiration for a number of latter approaches. Some drawbacks of REINFORCE are that it is on-policy, it is rather slow and it uses a naive way of sampling for the estimation problem. Sutton *et al.* (2000) show that, in contrast to REINFORCE, an approximate action-value function can be used to aid the estimation problem and they prove

⁴⁰Note that this is the general definition, based on Q -values.

convergence to a locally optimal policy. Meuleau and Peshkin (1999) provide *off-policy* implementations of REINFORCE by using *importance sampling* such that any well-behaved exploration policy can be used. The VAPS algorithm (Baird, 1999; Baird and Moore, 1999) does not use the gradient directly, but instead uses a combination of the performance and value function accuracy. In this way, a whole family of algorithms is obtained, ranging from pure value function search to policy search. Most methods focus on model-free RL. An exception is the *model-based* PG algorithm by Wang and Dietterich (2003) which estimates *partial models* that can be used to compute the gradient instead of MC sampling as in most PG methods, thereby reducing variance. A hierarchical extension by Ghavamzadeh and Mahadevan (2003) uses *value-based* RL for higher level subtasks, which are usually of lower complexity, and PG for lower level subtasks. Most methods use MLPs as FAs, but see (Bruske *et al.*, 1997) for an example of PG using *dynamic cell structures*.

Evolutionary Policy Search. A third way of obtaining policies directly is using *search*, usually by using *evolutionary algorithms* (EA) (Moriarty *et al.*, 1999) (see Section 3.6.2.2 for general references to the EA literature and Chapter 5 for an extended example in relational domains). A strong assumption of PG techniques is that a gradient can be computed. Many policy representations do not have a natural gradient defined, e.g. *rule-based* policies. EAs search directly in policy space, using a *fitness measure* to guide the search. The fitness of a policy is given by a single *scalar*, for example the total or average reward obtained by the policy in a set of test trials in the environment.

EAs for policy search can work on *complete* policy structures (e.g. NNs), or on *parts* of policy structures (e.g. *individual rules*). Examples of the second kind are *learning classifier systems* (LCS) (Lanzi *et al.*, 2000) where each *classifier* (i.e. condition-action rule) is associated with a value that represents its *strength*. Learning values for classifiers uses RL-type algorithms, whereas the structure of the classifiers is learned by the EA. In some restricted cases (e.g. when no generalization is used) *Q*-learning and LCSs can be shown equivalent (Dorigo and Bersini, 1994). Related to LCSs, Baum (1999) introduced the HAYEK machines, in which an *artificial economy* of agents, i.e. condition-action rules, *bid* on the right to suggest actions. Agents get rewarded relative to their quality and bids. An evolutionary algorithm searches for agents, and the approach showed impressive results in large BLOCKS WORLDS . In Section 3.6.2.2 we discussed several approaches (e.g. SANE Moriarty and Miikkulainen, 1996) that use EAs to find NNs. Evolutionary policy search has proved very useful for learning reactive behaviors in robotics (Dorigo and Colombetti, 1997; Nolfi and Floreano, 2000). Some disadvantages of EA are that they are less suitable for online learning and that their global behavior tends to neglect *rare* states, which can be a problem depending on the importance of these states. See (Taylor *et al.*, 2006) for some further comparisons between evolutionary and standard RL. Furthermore, theoretical advances in the field of EA is still limited, and convergence – either to a locally optimal or globally optimal – cannot be guaranteed in general.

3.8. ABSTRACTION TYPE V: Hierarchical and Temporal Abstraction

A currently very active subfield in RL is *hierarchical reinforcement learning* (HRL) (see Dietterich, 2000b; Barto and Mahadevan, 2003; Ryan, 2004a, for recent overviews). HRL consists of a whole range of abstraction methods that try to *decompose* an MDP into a *hierarchical* structure in which several *subproblems* can be solved more or less independently.

The basic strategy in HRL can be characterized as *divide and conquer*, introducing multi-level hierarchical decompositions where each level acts as gating mechanism switching between sub-controllers or behaviors in a similar manner in which programs call subroutines. Inspiration for HRL came from *behavior-based* approaches in *robotics* (see Arkin, 1998) and from the work in *hierarchical planning* and *hierarchical task networks* (see Erol *et al.*, 1994; Russell and Norvig, 2003). Hierarchical approaches focus on *temporal* and *task* abstractions. Temporal abstractions treat *sequences* of actions as one abstract, *temporally extended*, action. Such an action implements a *closed-loop, partial* policy that is generally defined for a *subset* of the state space. These partial policies are sometimes called *macro actions*, *macros*, *behaviors*, *skills* (Thrun and Schwartz, 1995), or *options* (Sutton *et al.*, 1999). We will use the term *behavior* in this section, and *primitive actions* for the one-step actions from the flat MDP definition. Behaviors can be structured in *task hierarchies* which can be seen as a strong bias on the policy space.

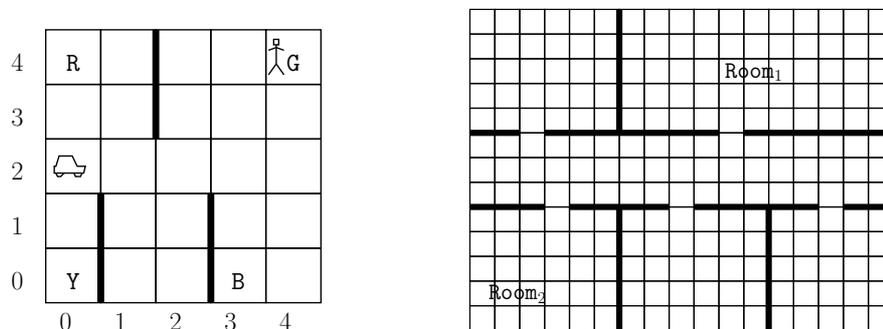


Figure 3.17: Two typical HRL domains. a) The taxi domain. b) A grid world with rooms.

Two example domains often used in these contexts are depicted in Figure 3.17. The *taxi domain* is a simple grid world that contains a *taxi*, a *passenger* and four designated locations (R, G, B and Y). At the beginning of an episode, the taxi is at a randomly selected grid position, and the passenger is at one of the four special locations. The passenger has a desired location where he wants to go and the task of the taxi is to get the passenger, pick him up, bring him to his desired location and drop him off. The taxi has 6 actions, which are actions for moving (north, south, east and west) and for handling the passenger (pickup and putdown). Actions can be deterministic or stochastic, and in some extensions, the world includes a *fuel point* and the taxi has an additional goal to *refuel* when needed. A high-level *task structure* (see also Figure 3.18) can represent that the taxi first has to navigate to the passenger, pick him up, navigate to the target location and deliver the passenger, though all these behaviors will consist of *sequences* of *primitive actions*. Note that the behavior of navigating to the passenger can be *reused* for navigating to the target location.

The second example is an instance of a *room world* in which the task of the agent is to get from room to room using primitive actions for moving as in the taxi domain. Low-level behavior will get the agent from one grid position to another, and on an abstract level the goal is to get from room to room. A *behavior* for traveling between rooms can be used, regardless of how it is implemented, e.g. of which individual move actions it consists. Such behaviors can be learned by simple *Q-learning*, using one room as an initial state and the target room as a sub-goal. These behaviors (i.e. partial policies over a sub-MDP) are easier

to learn, and when additionally state abstraction techniques are used, they depend on less state features.

HRL accelerates learning by forcing a *structure* on the policies being learnt. Reactive mappings from states to actions are replaced by a hierarchy of behaviors. For MDPs the extension adds to the sets of admissible actions $a \in A(s)$ for $s \in S$ a *set of behaviors*, each of which can itself invoke other behaviors, thus allowing a hierarchical specification of an overall policy. This also helps with the initial exploration of most RL algorithms, because the hierarchy restricts (random) action choices to actions that comply with the hierarchy. Most HRL methods work in a model-free setting. In general, the focus is not so much on globally optimal policies, but more on good policies that comply with the hierarchical abstraction level, allowing for modular structure, reuse of partial policies and faster learning. Following the PIAGET-levels, several HRL dimensions can be distinguished. Methods using fixed hierarchies (e.g. PIAGET-1 and PIAGET-2) are supplied with a *given* hierarchy and learn *behavior value functions* and *partial policies*. Model-minimization techniques for HRL (PIAGET-0) will be discussed in Section 3.8.3. Current research focuses mainly on algorithms that construct the hierarchy in an automated fashion (PIAGET-3) and we will discuss them in Section 3.8.4.

3.8.1 Semi-Markov Decision Processes

One of the differences between standard and hierarchical RL is that behaviors can be *temporally extended* whereas primitive action cannot. Executing a behavior will produce a sequence of transitions between states, and consequently a sequence of rewards. The *semi-Markov decision process* (SMDP) (see Puterman, 1994) is an extension of the MDP framework to accommodate action *duration*, or *multi-step actions*:

DEFINITION 3.8.1 ► A **semi-Markov decision process** (SMDP) M is a tuple $\langle S, B, T, R \rangle$ where S is a (finite) set of states, B a (finite) set of behaviors (i.e. temporally extended and primitive actions), $T : S \times B \times S \times \mathbb{N}^+ \rightarrow [0, 1]$ a transition function that includes the duration of behaviors and $R : S \times B \times \mathbb{R} \rightarrow [0, 1]$ a reward function distribution. Furthermore T and R are defined as:

$$T(a, B, s', n) = P(B_t \text{ terminates in } s' \text{ at time } (t + n) \mid s_t = s, B_t = B)$$

$$R(s, B, r) = P\left(\sum_{i=0}^{k-1} \gamma^i r_{t+i} = r \mid s_t = s, B_t = B\right)$$

B_t is the behavior at time t . T and R must both obey the Markov property, i.e. they can only depend on the behavior and the state in which it was started.

A policy π is a mapping from states to behaviors, i.e. $\pi : S \rightarrow B$. The execution of a behavior will result in a sequence of primitive actions with corresponding rewards. If a behavior B at time t starts in state s_t and terminates after k time steps in state s_{t+k} then the SMDP reward obtained is equal to the accumulation of the one-step rewards during the execution of B i.e. $R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^{k-1} r_{t+k-1}$. Standard Q -learning learns a state-action value function whereas SMDP Q -learning learns a *state-behavior value*

function Q^* defined as:

$$Q^*(s, B) = E \left[\sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V^*(s_{t+k}) \mid \epsilon(s, B, t) \right] \quad (3.5)$$

where $\epsilon(s, B, t)$ indicates that at time t the behavior B is executed in state s , and k is a random variable denoting the duration of behavior B . Since the value function for a behavior-based policy is identical to the value function for a primitive policy, we know that π^* is the optimal policy in the limited set of policies allowed by the behavior hierarchy.

Learning internal policies of behaviors can be expressed in a similar way. Let π_B the internal policy of behavior B and let $A(B)$ be the set of available sub-actions for behavior B (note that these can be either primitive actions or other behaviors). Let Main be the root behavior, with reward function equal to the original MDP. Two different forms of optimality can be defined. A *recursively optimal* policy has:

$$\pi_B^*(s) = \arg \max_{a \in A(B)} Q_B^*(s, a)$$

where $Q_B^*(s, a)$ is the optimal state-action value function for each behavior B according to its local reward function R_B . This local reward function is assumed to be in line with the goals of the behavior. Thus, each behavior is optimal. A *hierarchically optimal* policy has:

$$\pi_B^*(\text{stack}, s) = \arg \max_{a \in A(B)} Q_{\text{Main}}(\text{stack}, s, a)$$

where $\text{stack} = \{\text{Main}, \dots, B\}$ is the calling stack of behaviors and Q_{Main}^* is the state-action value function according to the root reward function. The actual best action is determined by the calling stack of behaviors; a behavior may behave differently depending on the *context* in which it is evoked. The functions R_B and Q_B^* are not defined as hierarchically optimal policies do not allow for local goals of behaviors. Most HRL methods focus on learning recursively optimal policies, because they allow for more flexible learning as behaviors more independent using this notion of optimality.

3.8.2 Fixed Hierarchical Abstractions

The field of HRL is very active and among the most well-known examples are MAXQ, HAM and the *options* framework and we discuss these in this section in some detail. Predecessors of hierarchical methods were proposed in the early nineties and most of this work is concerned with state space aggregations. *Feudal Q-learning* (Dayan and Hinton, 1993; Dayan, 1998) is a form of state aggregation applied to navigational problems. States are evenly aggregated according to spatial proximity (w.r.t. an underlying topology) and these aggregations are hierarchically aggregated to form a hierarchy of abstraction levels. A *manager* has the responsibility over an aggregation at some level in the hierarchy. Each manager is subordinate to a manager at the level above, creating a feudal hierarchy of independent learners. All but the lowest managers take actions by giving control to a sub-manager to achieve some subgoal. The managers at the lowest level perform actions in the flat MDP. Each manager gets rewards from its manager when it reaches some subgoal, and after that returns control to its manager. In this way, each manager learns to chose

which sub-manager should take an action in the abstract states over which it has responsibility. The HDG algorithm (Kaelbling, 1993a) solves large navigation tasks by distributing *landmarks* over the space to be navigated, and aggregating states according to their closest landmark. Learning paths between landmarks is much less complex than learning paths between arbitrary states. Early model-based approaches are *H-DYNA* (Singh, 1992), which is an extension of the DYNA approach (see Section 2.6.3) to deal with various levels of temporal abstractions, and the decomposition approach by Dean and Lin (1995), which computes *hierarchically optimal* policies, as does the HAM approach. *H-DYNA* and HDG both use a decomposition of the value function in addition to a decomposition of the policy, inspiring the work on MAXQ in this respect.

In the next paragraphs we discuss some well-known methods that show characteristics reoccurring in many currently proposed methods. Especially the *options* framework is used often in approaches that aim at learning *subgoals* and *behaviors* for reaching these subgoals. Most methods are model-free, but those that learn the hierarchy itself usually estimate a model to induce subgoals and behaviors. Other methods, such as that by Jonsson and Barto (2005) and Littman *et al.* (2005), assume a factored DBN model to be present.

Options and SMDP Q -learning. The simplest algorithmic framework extends standard Q -learning to include temporally abstract behaviors with hard-coded internal policies. Such a framework is the *options* framework introduced by Sutton *et al.* (1999). An option $\langle I, \pi, \beta \rangle$ consists of a (deterministic or probabilistic) policy $\pi : S \times \cup_{s \in S} A(s) \rightarrow [0, 1]$, a *termination condition* $\beta : S \rightarrow [0, 1]$ and an input set $I \subseteq S$. The option is available in a state s only when $s \in I$. If the option is executed, then actions are selected according to π until the option terminates stochastically according to β . Usually, if an option can continue in some state s , it can also be initiated in s , i.e. $\{s : \beta(s) < 1\} \subseteq I$. Any action a of the original MDP, i.e. a primitive action, is also an option with $I = \{s : a \in A(s)\}$ and $\beta(s) = 1$ for all $s \in S$. In effect, the original action space is *extended* with options, such that care has to be taken such that the options *do* in fact, speed-up learning in this enlarged state-action space⁴¹.

SMDP Q -learning learns *behavior value functions*, using standard Q -learning on the state-behavior space, discounting rewards obtained within behaviors in the proper way (see Section 3.8.1). The algorithms learns these value functions similar to standard Q -functions:

$$Q(s_t, B_t) = Q(s_t, B_t) + \alpha \left(R_t + \gamma^k \max_{B \in \mathcal{B}} Q(s_t, B) - Q(s_t, B_t) \right) \quad (3.6)$$

If the internal policies obey the Markov property, then such behaviors are semi-Markov and can be used in SMDP Q -learning. Other model-free, and model-based DP approaches such as value iteration and Monte Carlo methods, can be applied to options in the SMDP model. SMDP Q -learning converges to the optimal behavior-based policy using assumptions similar to those for the convergence of standard (one-step) Q -learning.

⁴¹In fact, a similar exposition can be seen in ILP and other constructive learning settings. By providing background knowledge, the hypothesis language becomes more powerful, and more compact structures might – in principle – be learned. However, the searchable space becomes much larger, and proper bias should be provided in order to benefit from the added features. In ILP this is done by language and search bias, whereas in (H)RL this bias comes from a task hierarchy and admissible action sets.

Algorithm 6 Hierarchical Semi-Markov Q -Learning (HSMQ).

```

1: function HSMQ(state  $s_t$ , action  $a_t$ )
2: returns sequence of state transitions  $\{ \langle s_t, a_t, s_{t+1}, \dots \rangle \}$ 
3:
4: if  $a_t$  is primitive then
5:   execute action  $a_t$  and observe next state  $s_{t+1}$  and return  $\{ \langle s_t, a_t, s_{t+1} \rangle \}$ 
6: else
7:   sequence  $S := \{ \}$  and behavior  $B := a_t$ 
8:    $A_t := \text{TaskHierarchy}(s_t, B)$ 
9:   while  $B$  is not terminated do
10:    choose action  $a_t := \pi_B(s_t)$  from  $A_t$  (according to some exploration policy)
11:    sequence  $S' := \text{HSMQ}(s_t, a_t)$ 
12:     $k := 0$  and  $\text{totalReward} := 0$ 
13:    for each  $\langle s, a, s' \rangle \in S'$  do
14:       $\text{totalReward} := \text{totalReward} + \gamma^k R_B(s, a, s')$ 
15:       $k := k + 1$ 
16:    observe next state  $s_{t+k}$ 
17:     $A_{t+k} := \text{TaskHierarchy}(s_{t+k}, B)$ 
18:     $Q_B(s_t, a_t) := Q_B(s_t, a_t) + \alpha(\text{totalReward} + \gamma^k \max_{a \in A_{t+k}} Q_B(s_{t+k}, a))$ 
19:     $S := S + S'$  and  $t := t + k$ 
20:   return  $S$ 

```

HSMQ-learning. *Hierarchical semi-Markov Q -learning* (HSMQ) (see Dietterich, 2000b) is a recursively optimal learning algorithm that learns reactive, behavior-based policies. HSMQ can be seen as an extension of SMDP Q -learning in which the update rule in Equation 3.6 is recursively applied, with a local reward function, at each level in the hierarchy. This hierarchy is hand-coded and limits the set of actions for each particular occasion. HSMQ-learning converges to a recursively optimal policy under the same assumptions as SMDP Q -learning. Algorithm 6 shows HSMQ-learning in pseudo-code. Here, `TaskHierarchy` is a function that returns the available actions (primitive or behaviors) applicable by a particular behavior in a given state. Note that it is assumed here that the hierarchy is supplied by a designer of the system. We address the difficult problem of learning hierarchies in Section 3.8.4.

The SMDP Q -learning algorithm defines a flat structure in which fixed option policies can be used as if they were primitive actions. This amounts to PIAGET-1 learning, where values are learned for a fixed abstraction level. The HSMQ-learning algorithm extends this approach to PIAGET-2 by learning the *internal* parameters (i.e. behavior policy) too. Both these algorithms form the basis for many other (constructive) HRL methods. Most methods are based on identifying *subgoals* and learn separate behaviors for reaching these subgoals separately. These behaviors can then be used in SMDP Q -learning.

MAXQ. The MAXQ algorithm (Dietterich, 1998, 2000b,a) can be seen as an extension of the HSMQ algorithm. It relies on the theory of SMDPs but unlike e.g. the options framework, it does not rely on reducing the complete problem to a single SMDP. Instead a *hierarchical task decomposition* in behaviors is assumed to be given, and learning proceeds in a way similar to SMDP- or HSMQ-learning. MAXQ learns policies equivalent to HSMQ, but in addition, it uses a sophisticated *value function decomposition* to learn these more

efficiently. The value of a behavior in the context of its calling parent behavior can be decomposed into **i)** the reward expected while executing it and **ii)** the discounted reward of continuing to execute the parent task after it terminates. Let P be the parent behavior of B , then

$$Q_P(s, B) = I_P(s, B) + C_P(P, s, B)$$

where $I_P(s, B)$ is the expected total discounted reward that is received while executing behavior B from initial state s and $C_P(P, s, B)$ is the expected total reward of continuing to execute behavior P after B has terminated, discounted appropriately with respect to the time spent on executing behavior B .

The $I_P(s, B)$ can be recursively decomposed into I and C following

$$I_P(s, B) = \max_{a \in A_B} Q_P(s, a)$$

There are several advantages of this decomposition, specifically in learning recursively optimal Q -values. Both the I and C functions can be represented using different state space abstractions, allowing for sharing (and compactness) in the representation of the value function. See (Dietterich, 2000a) for more subtle details on this decomposition.

HAMQ. *Q*-learning with hierarchies of abstract machines (HAMQ) (Parr and Russell, 1998) learns hierarchically optimal policies, and uses devices resembling finite-state machines to implement behaviors. These machines include an *internal state* and this state determines the actions that are taken. Like the options framework, HAMs are based on the SMDP model, though the aim here is not to enlarge the action space with behaviors, but instead to *simplify* large MDPs by restricting the policy space. The core idea in HAM is that the policies for the original MDP are defined as *programs* which execute based on their own state as well as the state of the underlying MDP. There are several types of actions. There are primitive actions, actions that terminate the current behavior and return control to the calling behavior and actions that call other behaviors. Learning takes place only at *choice points* where a behavior must decide which of several internal transitions to make. These choice-points represent a trade-off between fully hard-coded policies and learned policies. All the finite state machines representing the behaviors are *compiled* into one machine where the learning takes place. Andre and Russell (2001) extended the framework to *programmable* HAMs, adding interrupts and the ability to pass parameters, among other things, and later extended the programming language to ALISP (Andre and Russell, 2002, see also Chapter 7).

Other Issues in HRL. This section has shown two types of hierarchical abstraction. The first is the use of temporally extended behaviors (such as *options*) to enhance the action set

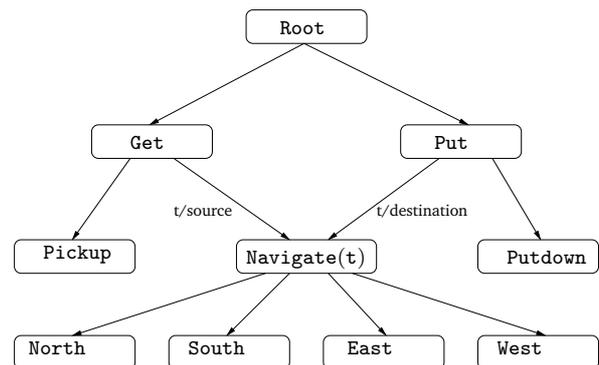


Figure 3.18: An example of *hierarchical abstraction*: the MAXQ task hierarchy (Dietterich, 2000a). The hierarchy constrains the policy space in the taxi domain. The leaves of the tree contain basic actions in the domain (such as North) and the inner nodes represent behaviors that are constructed using actions and behaviors lower in the tree. Note that the *Navigate* behavior is reused for both getting to the passenger and delivering him, and also that it is parameterized using the target location.

available to the agent. Behaviors can be seen as *shortcuts* to *subgoals*. The second type of hierarchical methods decompose the SMDP into a *task structure*. This represents a bias on the policy space. Hierarchical decompositions and behaviors help in two ways; they limit the choices available to the agent by placing a bias on the policy spaces and they allow for the specification of subgoals for parts of the policy. Additionally, they allow for different *state abstractions* to be used in different parts of the MDP. For example, in the taxi domain, the passenger's target location is not needed for the behavior that gets to the passenger. HRL methods have been combined with *policy gradient* techniques (e.g. Ghavamzadeh and Mahadevan, 2003, see also Section 3.7) and used in the context of POMDPs (e.g. Wiering and Schmidhuber, 1997; Theodorou, 2002). Furthermore, an extension of the EBRL approach (see Section 3.5.2) to HRL was made by Tadepalli and Dietterich (1997), and Littman *et al.* (2005) combined factored representations, MAXQ hierarchies and R-MAX (see Chapter 2) and proved polynomial bounds for by their algorithm to attain near-optimal performance within the hierarchy, with high probability.

There are two additional issues important in hierarchical abstractions. One is that of *termination* and *commitment*. Most methods assume that once a behavior is chosen, it is committed to first complete before returning control to the calling behavior. However, sometimes it is useful to allow for the termination of behaviors. For example, the taxi might stop navigating to the passenger when it senses that fuel is running out. Ryan (2002) describes algorithms where behaviors can be interrupted without affecting learning possibilities. A second issue in HRL is concerned with the fact that when behaviors are constructed from the same set of primitive actions, there may be a strong overlap between experiences gathered while executing various behaviors. This happens especially when behaviors have overlaps in the state space areas where they are applicable. There are two ways of doing this. One can update the *internal* policy for both behaviors (i.e. *all goals updating*), or one can update the *external* policy that is responsible for calling these behaviors (*intra-option* learning). The first has similarities with *off-policy* RL algorithms where samples generated by a different policy (the current behavior) are used to estimate values for another policy (another behavior). Both types of updating demand more computation per experience (higher computational complexity) with the advantage of possible faster convergence (lower sample complexity). Many more subtle issues are important for the combination of RL methods with temporal abstraction, see (Barto and Mahadevan, 2003; Ryan, 2004a) for more on these issues.

3.8.3 Model-Minimization for SMDPs

The model-minimization techniques in Section 3.4 show how to compute minimized homomorphisms of an MDP. Similar mechanisms can be shown to work for SMDPs too. The approach by Ravindran and Barto (2003a); Ravindran (2004) can be seen as an extension of the work on *stochastic bi-simulations* for MDPs (Givan *et al.*, 2003). It uses an *algebraic* approach to define MDP *homomorphisms* (Ravindran and Barto, 2001), and is related to earlier work on *machine homomorphisms* for *finite state automata*. Homomorphisms are conceptually simpler, and in addition the framework supports the exploitation of *symmetries*. Informally, homomorphisms here are mappings from one dynamical system to another that eliminate state distinctions while preserving the system's dynamics. Model-minimization for SMDPs implements PIAGET-0 for hierarchical abstractions, i.e. the aim is to compute abstractions before learning by providing a sound approach for identifying

structure in a given SMDP such that a more compact model can be induced (and solved instead of the original SMDP).

An SMDP *homomorphism* h is a set of surjections $\langle f, g_s \rangle$ from SMDP $M = \langle S, B, T, R \rangle$ to SMDP $M' = \langle S', B', T', R' \rangle$ with $h((s, b)) = (f(s), g_s(b))$, where $f : S \rightarrow S'$ and $g_s : B_s \rightarrow B'_{f(s)}$ for $s \in S$ such that $\forall s, s' \in S, a \in B_s$ and $N \in \mathbb{N}$

$$T'(f(s), g_s(b), f(s'), N) = \sum_{s'' \in [s']_f} T(s, b, s'', N)$$

$$R'(f(s), g_s(b), N) = R(s, b, N)$$

Note that an MDP homomorphism is similar to this definition, except that then these conditions can neglect transition times. The surjection f induces equivalence classes of states s of M (i.e. $[s]_f$). $R(s, b, N)$ is the expected reward for performing action b in state s and completing it in N steps. Each surjection g_s recodes the actions admissible in state s of M to actions admissible in state $f(s)$ of M' . This *state dependent recoding* of actions is the key component of the approach. It enables the exploitation of *symmetric equivalence* as a special case of homomorphic equivalence. Related work by Asadi and Huber (2005) first identifies subgoals and learns behaviors for reaching them off-line (see next section). After that, it computes an abstract SMDP by computing *action dependent partitions* based on *stability criteria* for minimizing MDP (see Section 3.4), adapted for the SMDP setting.

Often the homomorphic properties do not hold for the entire state space. Ravindran and Barto (2002) introduce the notion of *relativized options* based on *partial homomorphisms*, such that options can be defined without an absolute frame of reference. This opens up new possibilities for using the option, knowledge transfer and more efficient learning. Another way to loosen the constraints on exact homomorphisms is the definition of *approximate homomorphisms* in which *intervals* for transition probabilities and rewards are taken instead of exact preservation in the homomorphism. An efficient implementation of these ideas can be given for *factored* (see Section 3.5) representations of MDPs (Ravindran and Barto, 2003b).

3.8.4 Dynamic Hierarchical Abstractions

In the previous sections we have highlighted some of the principles of HRL and discussed some of the prominent methods. However, in all these methods the hierarchical abstraction level is fixed and usually supplied as a bias. Model-minimization implements PIAGET-0, methods such as options implement PIAGET-1 and HSMQ and MAXQ implement PIAGET-2. A suitable set of behaviors can help improve the agent's efficiency in learning to solve difficult problems. If an agent can develop such behaviors automatically, it should be able to efficiently solve a variety of problems without relying on hand-coded behaviors tailored to specific problems. In general we can distinguish two main directions. One is that of identifying useful *subgoals* and learning behaviors for reaching them, usually embedded in the *options* framework (Sutton *et al.*, 1999). The second direction – which is more difficult and less well developed so far – is to learn a complete *task hierarchy*.

Subgoals are generally considered to be *bottlenecks* in the state space, e.g. a passage between two rooms in the environment. They are generally believed to be of strategic importance and worthwhile reaching. It is useful to identify these states, such that efficient behaviors can be learned in isolation and *exploited* for learning the complete task. An early

approach is the SKILLS algorithm by Thrun and Schwartz (1995). They do not explicitly talk about subgoal discovery, but instead they study identifying commonly occurring sub-policies, i.e. *skills*, in solutions to a set of tasks. These skills are then *compacted* (e.g. by using an MDL measure) and can then be reused. The related POLICYBLOCKS approach by Pickett and Barto (2002) too finds commonalities in solutions to a set of tasks but defines a sophisticated mechanism to remove options that are already "explained" by other options.

An early approach based on subgoal identification is HQ-learning (Wiering and Schmidhuber, 1997). It was designed for (deterministic) POMDPs. Tasks are solved by reactive policies (i.e. agents) that pass *control* over the task to each other after they have reached their own subgoal. At the end of each episode agents adjust their policy and subgoals based on rewards. A related approach is HASSLE by Bakker and Schmidhuber (2004) which additionally uses state abstractions. Both methods have a two-layer approach in which the higher level is in charge which subgoals to reach and the lower level policies learn how to reach subgoals. Both methods are also quite flexible in the way the task is *segmented* into several subproblems to be solved by the lower-level policies.

Most recent methods first *identify* subgoals, then *learn an option* for that subgoal, and then *incorporate* the new option in the agent's action set. The main difference in subgoal discovery methods is the definition of *what* is a subgoal. Digney (1998) chooses states as subgoals which have non-typical rewards, i.e. where the reward gradient is high. Mc-Govern and Barto (2001) choose states based on their frequency of appearance. The idea is that they are likely to be part of the agent's optimal path. The frequency is combined with a success criterium: states serve as potential subgoals if they are frequently visited on successful paths but are not visited on unsuccessful paths. One problem is that much exploration is needed for identifying subgoals such that they are discovered at relatively advanced stages in learning. Simsek and Barto (2004)'s *relative novelty* (RN) approach is based on finding *access states* that allow the agent to transition to a part of the state space that is otherwise unavailable or difficult to reach from its current region. This type of subgoals can be identified without solving the complete task, and they can often be transferred to tasks with different reward functions, i.e. getting through a door is useful independent of what the agent should do in the room. Access states will be more likely than other states to introduce short-term novelty, i.e. mediate a transition to a region not visited recently. Asadi and Huber (2005) learn subgoals by analyzing the learned policy by MC sampling to identify states that lie on a substantially larger number of paths than would be expected by looking at their successor states.

A number of related approaches uses the *state transition graph* to identify subgoals. The Q-CUT algorithm (Menache *et al.*, 2002) uses *bottleneck* states which separate well initial and target states. Menache *et al.* define subgoals as border states of strongly connected areas of the MDP transition graph and find them using max-flow/min-cut algorithm thereby finding a *cut* of the graph to separate the graph as much as possible. Mannor *et al.* (2004) consider *clusters* in the state graph as intermediate stages. Subgoals are *sets* of states, as opposed by many other methods that use single states as subgoals. Recorded state transitions, and additionally the value function, are used to find clusters in the state graph. Measures of *intra-cluster* quality and *inter-cluster* quality are used to find a good clustering of states after which options are learned that take the agent from a cluster to neighboring ones. Clustering starts each time when no new states are encountered for some while. (Simsek *et al.*, 2005) take a similar graph-theoretic approach, now based on *access states*,

but find cuts in *local* state graphs instead of the mentioned *global* approaches. This *L-CUT* algorithm, as does RN, works on the most recent part of the transition history. The algorithm finds an optimal cut in the transition graph for this history and if the cut is classified as a subgoal (using a probabilistic measure of the likelihood of states being subgoals) an option is generated for this subgoal.

State abstraction can be an integral part of algorithms, such as the MAXQ hierarchies. Some methods specifically target the problem of generating *option specific* state abstractions. Jonsson and Barto (2001) adapt the UTREE algorithm (McCallum, 1996, see also Section 3.6.2.3) for use in a given hierarchy (using the *options* framework). Jong and Stone (2004) take a Bayesian approach to determining the relevance of a state variable for behaving optimally within an arbitrary region of the state space. These state abstractions can then form the basis for reusable tasks.

The second direction in *dynamic* abstractions in HRL is based on building policy hierarchies in an automated fashion. The HEXQ algorithm (Hengst, 2002) uses the state variables to construct the hierarchy, such that the maximum depth of the hierarchy is the same as the number of variables. The hierarchy is constructed *bottom-up*. The lowest level is associated with the variable that changes most frequently⁴² and this level is the only one that directly interacts with the environment through primitive actions. States at a particular level are partitioned into Markov regions, and the boundaries between regions are identified by *unpredictable*, i.e. non-Markov, transitions which are called *exit states*. Sub-policies are then constructed to leave each region through the exit states and these sub-policies can then be used higher up in the hierarchy. Several extensions to HEXQ have been proposed. Hengst (2003) introduce *safe*⁴³ *state abstractions* for HEXQ and additionally the use of discounting and infinite-horizon using state abstractions. Hengst (2004) introduces three ways to *approximate* the hierarchy, trading optimality for space complexity. The extensions are: **i)** a limited search depth in calculating values which decreases the execution time, **ii)** combination of exit states of sub-MDPs which decreases both execution time and storage requirements and **iii)** combination of exits (i.e. actions) which leads to less storage and increased execution speed. The original HEXQ definition requires a period of initial exploration but Potts and Hengst (2004a,b) describe an approach to discover multiple levels of the hierarchy simultaneously, similar in spirit to the HSMQ algorithm, though now in a constructive approach.

Two more recent methods have their roots in graphical models. The work by Manfredi and Mahadevan (2005) uses a new type of graphical model, *dynamic abstraction networks* (DAN), that combine both state and temporal abstractions. All levels of the state and policy hierarchies, including possibilities for hidden state variables, are learned simultaneously through joint inference on the model. For training, supervised EM (Dempster *et al.*, 1977) is used on sequences of state-action pairs generated by a *guidance* policy supplied by the designer. From the generated DAN, policies can be obtained and improved using more traditional HRL algorithms, in this case the options framework. The *variable influence structure analysis* (VISA) approach by Jonsson and Barto (2005) uses a DBN factored representation (see Section 3.5), i.e. causal relationships between state variables, to decompose an MDP. Similar to HEXQ, VISA identifies *exits* that cause the value of state

⁴²The rationale behind this is that X and Y locations of a robot will change almost every step, whereas a *room location variable* will only change every now and then.

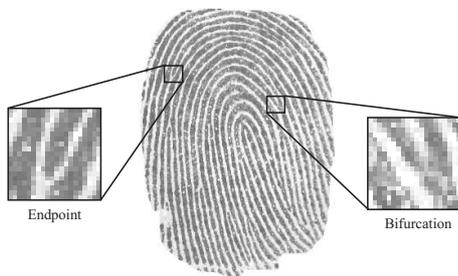
⁴³Meaning that the value function over all states is the same before and after abstraction.

variables to change. For each exit, VISA uses tree manipulations to construct an associated exit option. VISA learns recursively optimal policies. Compared to approaches such as SPI (see Boutilier *et al.*, 2000a, and Section 3.5) and sRTDP (Feng *et al.*, 2003), VISA showed experimentally a significant speedup for large problems. Research on the automatic discovery of hierarchical abstractions is still very relevant and active (e.g. see Mehta *et al.*, 2008, on the automatic discovery of MAXQ hierarchies).

3.9. An Abstraction Case Study: Fingerprint Recognition

Let us now briefly review a concrete application of RL to a real-world problem; the automatic recognition of fingerprint images. This example shows some specific abstraction and generalization techniques. Furthermore, it is an interesting domain because it is somewhat outside typical RL applications such as games and agent navigation.

One of the problems in *automated fingerprint recognition* is the robust *extraction*



of *minutiae* from a fingerprint image. In this section, we describe an application of RL with VFA to this problem (see Bazen *et al.*, 2001). In Figure 3.19, a fingerprint is depicted. The information carrying features in a fingerprint are the line structures, called *ridges* and *valleys*. In this figure, the ridges are black and the valleys are white. *Minutiae* are ridge-endings and bifurcations. They provide the details of the ridge-valley structures. Minutiae are used for fingerprint *matching*, which is a one-to-one comparison of two fingerprints⁴⁴.

Figure 3.19: Example of a *fingerprint* and two *minutiae*.

One of the first steps in fingerprint recognition is the extraction of the minutiae from the fingerprint image.

The classical approach uses a number of image processing steps for this task (Jain *et al.*, 1997). First, the fingerprint image is filtered for noise suppression. Then, a threshold is applied to the image in order to obtain a binary image. Next, the ridges are thinned to 1 pixel width by morphological operations. From this skeleton, minutiae extraction is a straightforward task. However, this method is highly sensitive to noise and bad image quality. This results in the extraction of many false minutiae. Therefore other, more robust, methods have to be investigated. All these operations perform feature engineering and feature selection to finally arrive at the level of features that can be used in the RL approach that we describe here.

RL can be used to control an image-exploring agent to extract the minutiae from a fingerprint image. It has been shown that it is good practice to follow the ridges in the fingerprint until a minutia is found. Maio and Maltoni (1997) presented an agent that takes small steps along the ridge. Jiang *et al.* (2001) enhanced the agent by using a variable step size and a directional filter for noise suppression. This results in a rather

⁴⁴In contrary to what is shown in popular television series such as *Crime Scene Investigation* and movies, the *minutiae* carry the characteristic features of a person's fingerprint. Often one sees the detective using a computer to find the *exact* picture of the criminal in a database. To prove that they have found the correct person, they often show two fingerprints moved on top of each other and all the lines coincide. Of course, the picture of the suspect's fingerprint taken from the last time the person was arrested and the picture taken at the crime scene will not be *exactly* the same. Instead, one has to verify that the minutiae of both prints have the same relative structure in both, with respect to the ridges and valleys.

complex system, especially since robustness is required. A much simpler solution is to use an agent that *learns* the task. By using a variety of training examples, a robust system can be obtained. van der Meulen *et al.* (2001) used *genetic programming* to evolve a minutiae extracting agent. In the approach of this section, the agent is trained by means of a standard Q -learning approach where the value is approximated using a neural network.

3.9.1 Reinforcement Learning for Minutiae Detection

The *goal* of the agent is to follow a ridge and stop at a minutia (see Figure 3.19). Furthermore, the minutia should be found in as few steps as

possible in order to minimize the computational time needed for minutiae extraction. This is a typical example of an *episodic task*, which terminates when a minutia is found. The *state* of the agent consists of a local view of the fingerprint image around it. It can observe the gray scale pixel values in a segment of $n \times n$ (for instance 12×12) pixels around it. The *orientation* of the agent is normalized by rotating the local view, such that the forward direction is always along the ridge-valley structures. For this purpose, the *directional field* is used (Bazen and Gerez, 2000). The local view is illustrated in Figure 3.20. This introduces considerable *a priori* knowledge. The agent is always aligned with the directional field, and this puts restrictions on the state space of the agent. Local views in which the agent has a direction other than approximately aligned with the ridges in the fingerprint do not occur.

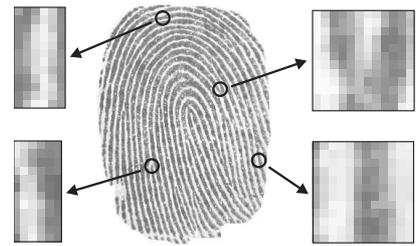


Figure 3.20: *Situatedness: a local view of the agent of 12×12 pixels.*

The length of the feature vector, which contains the pixel values in the local view, is reduced by a *Karhunen-Loève transformation* (KLT) Jain (1989). The KLT transforms feature vectors to a new basis that is given by the *eigenvectors* of their covariance matrix. Then, only those KL components that correspond to the largest eigenvalues are selected. This way, the largest amount of information is preserved in the smallest transformed feature vector. This has two useful effects. First, the length of the feature vector is reduced, which simplifies the learning process. Second, the KL components that carry the least information and therefore represent the noise, are discarded. This noise suppression method eliminates the need of a directional filter, which would require approximately 5 times as many computations.

The *actions* of the agent are the moves that it can make in the local view. Since the agent is always approximately aligned with respect to the directional field, the forward action is always a move approximately along the ridge-valley structures, and the agent does not need rotational or backward actions. To be able to achieve its goals, moving as fast as possible along the ridges and stopping at minutiae, the agent may move up to 4 pixels forward at each step and up to 1 pixel to the left or to the right. The left-right actions are necessary for keeping on the ridge. Although the directional field puts the agent in the right direction, it is not sufficient to keep the agent exactly on the ridges. The action results in a new position in the fingerprint, after which a new local view is extracted at that

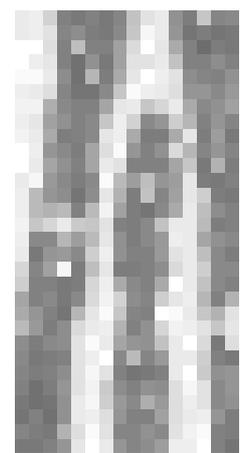


Figure 3.21: *A ridge with an endpoint.*

position.

The *reward structure* is a key element in this application. It determines the optimal actions for the agent to take. First, the agent receives rewards for staying at the center of a ridge. This is implemented by manually marking the ridge centers and using an exponential Gaussian function of the distance of the agent to the ridge center. Second, a much higher reward is given near the endpoints to be detected. Again a Gaussian function of the distance to the minutia is used. This structure encourages the agent to move in as few steps as possible to the endpoint by following a ridge and then to stop moving in order to receive the higher reward forever. This is enforced even more by scaling the reward such that large steps along the ridge receive a higher reward than small steps, while the opposite applies at endpoint locations. The reward structure around a ridge (see Figure 3.21) with an endpoint is shown in Figure 3.22. The peak in the reward corresponds to the endpoint of the ridge and the line of higher reward correspond to the ridge in the fingerprint.

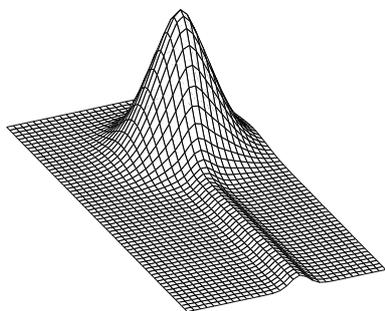


Figure 3.22: A reward structure.

Training works by selecting a ridge, initiating the agent at some location on that ridge, and defining the reward structure with respect to that ridge, including the target minutia. Then, the agent starts moving and the network is updated until the episode ends when the agent moves too far from the target ridge. It is worth noticing that during one episode the agent is trained to follow a certain ridge and it only gets rewards for being near that particular ridge. Therefore, the reward structure is different for each different ridge that is used for training. After each step in the environment the network's weights are updated. Examples can be obtained by

letting the agent interact with its environment, whereby the neural network determines the agent's actions. The training itself uses online adaptation of the NN.

During *testing* and actual *use*, agents are initiated at a large number of positions in the image, for instance on a regular grid. They start moving according to their policy and follow the ridges. However, there is no clear *termination* criterion for the agents, since it is not known in advance which minutia they should find. To overcome this problem, endpoints are detected by counting the number of successive small steps. After 5 small steps, the agent is terminated and an endpoint is detected. Bifurcations are detected by keeping a map of the trajectories of all agents. When an agent crosses the path of another one, the agent is terminated and a bifurcation is detected.

3.9.2 Experimental Results

The training is performed on the fingerprint image of Figure 3.19. In this fingerprint, all ridges and all minutiae have been marked manually. For each episode, one ridge is selected and the agent is initiated at a random position on that ridge. Then, the reward structure is calculated for that ridge as explained in Figure 3.22. Along the ridge, $\sigma^2 = 1$ and the amplitude is 1, while at a minutiae, $\sigma^2 = 10$ and the amplitude is 10. Next, the agent is trained by the SARSA algorithm as explained in the previous chapter. Finally, the agent is terminated if it is more than 7 pixels from the indicated ridge center. A multitude of experiments has been performed to find the optimal setup of the algorithm and here we describe some parameter settings that have been used. For one training session, 50k episodes were used.

During training, the exploration parameter decreases from $\epsilon = 0.01$ to $\epsilon = 0$. The agent's local view size was taken as 12×12 pixels, which was reduced to a feature vector of length 50 by the KL transformation. The actions were constrained to a grid of discrete values up to 1 pixel to the left or to the right and up to 4 pixels forward. The multi-layer perceptron had 1 hidden layer of 22 neurons and the learning rate was $\alpha = 10^{-2}$. The discount factor was set to $\gamma = 0.9$.

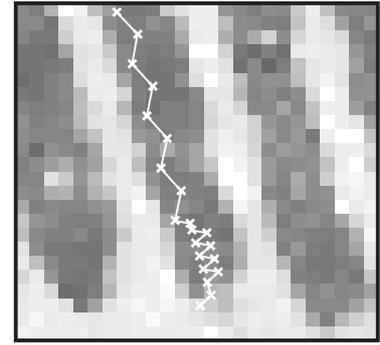


Figure 3.23: A *path along a ridge*.

Testing was performed on another fingerprint as described in Section 3.9.1. Starting points have been selected at a regular grid and the agents follow their policy until termination (see Figure 3.23). Human inspection of the results, shown in Figure 3.24, indicates that the agent follows the ridges and that all minutiae have been detected. The agent takes relatively large steps along the ridges, while the step size decreases near the endpoints. Furthermore, it intersects its own path at bifurcations. However, the figure also shows a number of false minutiae. These might be eliminated by further training of the agent on other fingerprints and fine-tuning of the parameters. Another possibility is the application of post-processing techniques to eliminate false minutiae structures.

3.9.3 Benefits of Various Abstractions

RL is a useful and intuitive way to tackle the problem of robust minutiae extraction from fingerprints. It has been shown that an adaptive agent can be trained to walk along the ridges in a fingerprint and mark minutiae when encountered. The use of VFA is necessary in these kinds of domains. When using a local view of 8×8 pixels, each of which can take values ranging from 0 to 255, the size of the state space amounts to $2^{14} \sim 16M$ states. The size of the state-action space is essentially infinite, because the actions are continuous (although we discretize in order to be able to choose a concrete action). Doing table-based Q in these kinds of state spaces is infeasible and furthermore, not needed. Distinguishing between two local views that only slightly differ in pixels values, is not needed.

In this application, several abstraction mechanisms are used. The first line of abstraction derives the actual data for this application. A fingerprint is obtained through a special sensor and its data is processed to generate a simplified representation stored in a database. Second, the stored fingerprint images are abstracted by taking only a local $n \times n$ view around the point where an action has to be taken. The third way abstraction is used is in the way the local views are transformed into state vectors. The KLT lowers the dimensionality of the state vectors describing the local view using a linear projection onto a lower-dimensional space. This abstraction is not lossless, because it throws away some information (from $12 \times 12 = 144$ to 50 features), but in return it also lowers the amount of noise in the state representation. A fourth line of abstraction is done for the action space. Actions are continuous inputs to the neural network and so the action space is essentially infinite. However, using a grid of discretized action values enables a fast, though not optimal, choice of actions. Thus the continuous action space is discretized into a finite grid of discrete actions. The final line of abstraction is done by the neural network. Instead of building a table of all reduced state vectors, the neural network is used to approximate the mapping from states (and actions) to values. The amount of tunable variables (e.g.

the weights of the neural network) is much lower than the number of states (from roughly $16M$ to about 1000). The complete abstraction process transforms the original, complex, noisy picture of a real fingerprint in several steps into a relatively small input to the neural network and finally mapped onto one real value.



Figure 3.24: *Extracted minutiae.*

In this particular application, the use of other techniques might be considered too. For example, the use of *value-based* RL algorithms can turn out to be not the best choice in this application. The similar local views on different places on the ridges creates a kind of *partial-observable* state representation, which did not create severe problems in this case though. It might be better to search directly for a policy by using *policy gradient* (Sutton *et al.*, 2000) methods or to use relative Q -values instead. Comparisons with different kinds of approximation architectures (in addition to neural networks) as well as other kinds of algorithms, such as $Q(\lambda)$ -learning might increase robustness and optimize learning time. In principle, the choice for other learning algorithms (such as policy gradient) might use the same abstraction procedure, whereas a different chain of abstractions (e.g. by using other kinds of features and linear transformations) might use the same RL algorithms.

3.10. Discussion

As we have seen, many choices exist for richer representations in the MDP framework to scale up to larger problems. This is possible by first representing the problem states using attribute-value features, and then by making use of various forms of structure in the problem and solution for abstraction and generalization purposes. Atomic representations as discussed in Chapter 2 do not allow for the exploitation of any kind of structure. Yet, *ignoring structure is the main cause of the curse of dimensionality*. In this chapter we have introduced the PIAGET principle in Section 3.3.2, that enhances GPI with abstraction and generalization, and we have distinguished five main types of abstraction in Sections 3.4 to 3.8. The core challenge for MDPs is to find smaller representations of the model (e.g. S , T , R) or solution (e.g. V , Q , π) such that fewer components have to be stored and retrieved, and such that learning experiences can be generalized to other, 'similar' situations. In some cases, these smaller representations are lossless, i.e. no real information is lost and solutions that are found are still optimal in the original problem. In other cases, solutions are approximated, i.e. some information is lost and solutions are only sub-optimal. This can happen when a small loss in performance is acceptable, but also when the problem is still too large and an approximated solution is the best one can do. In many cases when approximation is used, bounds on performance loss can be computed beforehand. We have seen many examples of situations where less-than-optimal performance was either tolerated or even traded off against speed of learning. The difference between hierarchical and recursive optimality, the accuracy of value function approximations, the non-Markov nature of state space aggregations and the heuristic fitness function evaluations in policy search methods, are all examples of approximations. The core problem when using ab-

straction and generalization, is *how to relate the values of the underlying structures to the abstractions used*. For example, in hierarchical solution techniques, the value of a sub-policy can be related to the cumulative values of the actions that make up the sub-policy. For more information about the role of representation, as well as the effects of abstraction techniques, see (Boutilier, 1999; Aler *et al.*, 2000; Finton, 2002; Koenig and Liu, 2002; Levner *et al.*, 2002)

Although we have described several dimensions and PIAGET levels in isolation, many *combinations* quite naturally arise. Concerning the PIAGET-levels, for example, model-minimization methods (PIAGET-0) are usually followed by a PIAGET-1 phase in which the compact, abstract MDP is solved. Combinations of dimensions occur often, in many systems. For example, the use of *state abstraction* is a common way to reduce the state-space in which behaviors operate, for example in MAXQ. In addition to methods that either implicitly or explicitly combine several types of abstractions, a number of works has explicitly combined or compared different kinds of abstraction. Littman *et al.* (2005) combine factored representations, MAXQ hierarchical abstraction and the R-MAX algorithm. They show that models can be combined with hierarchy without disrupting the benefits of factored states. Polynomial bounds on sample complexity of existing combinations with factored states are retained, where the hierarchy provides an approach to efficient planning in the learned models. Fitch *et al.* (2005) compare various kinds of abstraction in the TAXI domain, such as symmetry, homomorphisms and parallel (i.e. multi-agent) decompositions and show experimentally that the performance gains of single abstractions can be further improved through their combination. Whereas Fitch *et al.* quantitatively try to examine the benefits of a combination of some distinct approaches for learning, the approach by Steinkraus and Kaelbling (2004) considers *modeling* a hierarchy of abstraction modules, dealing with factored representations and temporal abstraction (OPTIONS). It improves on HAMs by taking a factored representation (which makes it easier to do local abstractions) and it can dynamically change the used representations.

Rethinking. When abstraction and generalization are introduced into the MDP framework, we are forced to *rethink* its components (see also Section 3.3.1). Rethinking state spaces is about recognizing that they are no longer just *identifiers*, but that they consist of features. Feature-based structure can be used to form descriptions that say which features are present in the current (abstract) state and which are not. This, in turn, creates clusters of states that fulfill the same descriptions, and by that *abstract states*. In the POMDP setting, these features can be aliased, i.e. they do not convey all the information about the state, whereas in this book we mainly focus on the fully-observable case (i.e. MDPs). Feature-based representations can also be used to introduce various forms of new structure, manifolds, and distances in the state space, and subsequently, basis functions, prototype states, and state aggregations, to capture various sources of structure in the original state space. This makes us also rethink value functions, which are now defined over prototype states, clusters, descriptions or basis functions. Approximation architectures such as neural networks and decision trees make use of these new state space definitions to assign values to complete regions in the state space, thereby generalizing over many individual states. Based on these properties of states, MDP definitions themselves need to be rethought. Because now, action definitions, reward functions, and transition functions can all use the feature-based nature of the states to exploit structure, for example in the definition of STRIPS action rules, or DBN specifications of transition probabilities. In general,

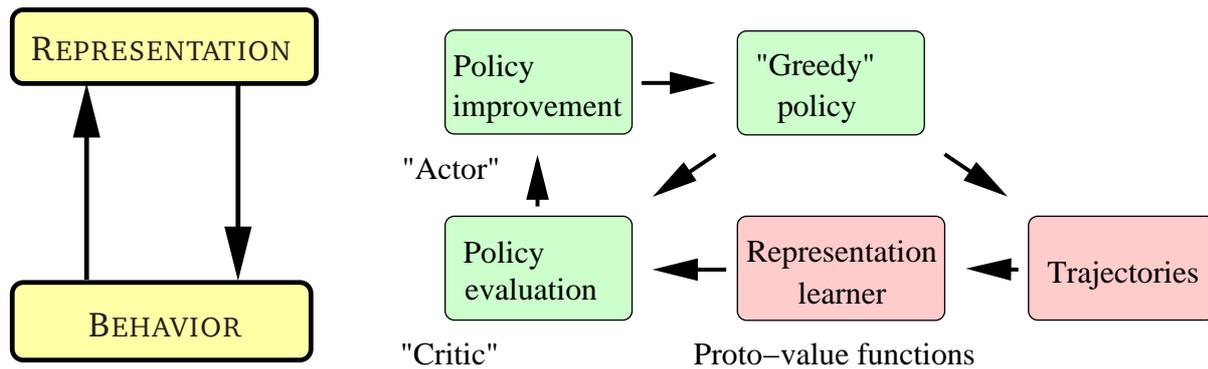


Figure 3.25: a) The interplay between representation and behavior. b) Representation Policy Iteration by Mahadevan (2005a).

feature-based representations introduce many opportunities to *induce* descriptions from experience. Rethinking actions results in the recognition that *abstract actions* can denote the simultaneous definition of multiple individual transitions in the original MDP using compact specification formalisms. But, as in hierarchical decompositions of MDPs, we may also see them as sequences of atomic actions. Either way, abstract actions provide higher-level control structures that can be used to transfer experience from one atomic action to the next, resulting in faster learning. As a consequence, rethinking actions leads to rethinking policies. That is because policies employ the same kind of abstraction and generalization mechanisms as in abstract actions. Furthermore, abstract policies can be defined in terms of abstract actions, and can put constraints on how (atomic or abstract) actions can be concatenated. Feature-based representations also provide opportunities to induce abstract policies from experience.

Learning Representation and Control. Going from GPI to PIAGET has been the most fundamental change we have described in this chapter. With the PIAGET principle, we basically distinguish between three types of algorithms. The first (PIAGET-0) is about obtaining a suitable representation of a problem. This is about feature construction, basis function construction, model-minimization, the construction of policy hierarchies and so on. In many cases this is considered a *pre-phase*, often entirely setup by a designer of the system, though in some cases these types of computation can be automated by learning. The second (PIAGET-1 and PIAGET-2) is about MDP solution algorithms (as in standard GPI), but now using abstraction and approximation architectures. A standard example is the use of a value function approximation architecture such as a neural network in Q -learning. The distinction between PIAGET-1 and PIAGET-2 is about whether the generalization architecture possesses its own parameters or not. The third type (PIAGET-3) combines the effect of the first two types and is about algorithms that learn both their representation and parameters in a single algorithm. These are the types of algorithms that are typically interesting for RL because they can function without much prior knowledge about the domain. One could define a fifth PIAGET level, characterizing the phase *after* learning relating to a recent direction in ML that is occupied with *transfer* of solutions to other, related problems. Suitable abstraction levels, general skills and induced policies might apply to other instances of the same problem, or to related problems displaying the same structure. This direction is beyond the scope of this book, though we will encounter some examples in the remaining chapters.

Figure 3.25a shows a very high-level, conceptual picture of the interplay between *representation* and *behavior* (as in PIAGET-3). Systems that can learn their own representation of the problem, are faced with a complex problem. The current representation of the problem drives the experience generation from which the system can learn how to behave. But at the same time, the experience that is generated, drives the modification of the representation. In other words, our current view of how the world looks like, influences the things we can observe, but at the same time, the things we can observe, influence how we can generalize. This is very related to POMDPs where the observations are influenced by the way we can see the world. However, in POMDPs there are limits on what we can observe, whereas when using abstraction in MDPs we *choose* not to pay attention to certain state features. Note the figure basically corresponds to PIAGET-3 which is the only type of learning making the full circle. The convergence of various types of algorithms can now be studied at two different levels. One is the standard convergence to a stable (or optimal) value function or policy. The other is the convergence to a stable (or adequate) representation. This also offers possibilities to approximate in the *algorithmic* part (e.g. the experience generation) or in the *representational* part (e.g. in building approximate structures for representation). We defer a discussion of these matters to Section 4.5.2.2 in the next chapter.

The problem of learning both representation and behavior appears in many contexts. In the beginning of this chapter we have mentioned *constructivism* in psychology (Piaget, 1950) and AI (Drescher, 1991), and we have discussed *representation change* and ways to characterize this process (Korf, 1980; Saitta, 1996). Throughout the chapter we have seen quite a number of approaches that employ constructivist methods in particular formalisms, or for particular MDP solution algorithms. Fewer texts deal with the general interplay between representation learning and control learning. Several recent PhD theses deal on in a more integral way with these matters such as in the *cognitive economy framework* (Finton, 2002), the factored MDP framework (Dearden, 2000), and using constructive neural networks (Großmann, 2001). In general ML literature it is sometimes stated as the distinction between *static* and *dynamic* generalization, in the neural networks literature the problem is known under the name *stability-plasticity dilemma*, in the RL literature it is related to the exploration-exploitation dilemma (see Chapter 2) and furthermore it is related to the *epistemological versus heuristic distinction* made by McCarthy and Hayes (1969) and the *declarative versus procedural distinction* described by Dietterich (2003). In ML for large datasets a similar problem is tackled, where feature (i.e. representation) learning goes hand in hand with example generation (e.g. see Blum and Langley, 1997).

In the more restricted context of sequential decision making under uncertainty, we have introduced our novel PIAGET principle, where the representation part includes aspects ranging from abstract state spaces to policy structures. The control part in PIAGET is about learning values, and about learning actions that maximize rewards. Earlier, Scott and Markovitch (1991) discussed similar issues in the DIDO architecture. They distinguish between two interactive processes in learning. One process generates suitable representations for the architecture based on experience. The other process is responsible for generating experience. An interesting interplay exists between the first process that tries to minimize the amount of uncertainty (e.g. exploitation) whereas the second tries to find those examples that are most informative, i.e. that maximize uncertainty and thus present new experience (e.g. exploration). Whether an example is informative depends on its

characteristics, the current state of the learning system’s representation of the domain and on the methods that are employed to modify this representation.

We have discussed several examples in the MDP literature that build more or less general representations that fit this same pattern, in the form of *basis functions* (Wu and Givan, 2005; Keller *et al.*, 2006; Parr *et al.*, 2007). These methods construct and refine their representation of the domain, for example by looking at Bellman residual errors that can be computed from experience that is generated in the domain. A recent method that promises to be a more general method fitting the pattern, is proposed by Mahadevan and Maggioni (2007, see also (Mahadevan, 2005a,b)). It automatically generating subspaces on which to project the value function using spectral analysis of operators on graphs, and it differs fundamentally from other attempts at basis function generation, for example tuning the parameters of pre-defined basis functions, dynamically allocating new parametric basis functions based on state space trajectories or generating basis functions using the Bellman error in approximating a specific value function. The method constructs manifolds, which are low-dimensional representations of objects in high-dimensional spaces and can be used to compactly represent the state space of a task. Basis functions are constructed from spectral analysis of diffusion operators where the resulting representations are constructed *without explicitly taking rewards into account*. The representational framework (involving so-called *proto-value functions*) is used in *Representation policy iteration* (RPI) (Mahadevan, 2005b), which alternates between a *representation step*, in which the manifold representation is improved given the current policy, and a *policy step*, in which the policy is improved, given the current representation, see Figure 3.25b. Proto-value functions have been used in continuous and hierarchical RL, though not yet in relational domains.

Abstraction and Generalization go Relational. Starting from the next chapter, we turn to *first-order* domains, consisting of *objects* and *relations*. The most fundamental change will be that states and actions have a richer structure. Interestingly, we will see that first-order approaches in MDPs are essentially *first-order upgrades* of the methods described in this and the previous chapter. In the next chapter we will first upgrade the representational formalisms, the ML methodology and action formalisms to the first-order case. Once that is done, all five types of abstraction that we have found in the current chapter can be systematically upgraded to work with first-order representations. In addition, we will see that the same four PIAGET-levels will be distinguished in this new setting. In Table 3.1 we sum up the five types of abstraction, the PIAGET-levels, which types of parameters and which types of structures are being learned, and the parts of this book in which first-order versions of the algorithms will be presented.

Approximation Type	Structures	Main Chapters	PIAGET-0	PIAGET-1	PIAGET-2	PIAGET-3
State Space Aggregation	state aggregation, distances, kernels, features, basis functions	Ch. 4-Ch. 7	§ 5.3.1	CARCASS § 5.2		§ 5.3.1.1
Factored MDPs	Structured V, Q, T, R, π	Ch. 4, Ch. 6 § 4.5.1.1, § 5.3.1		§ 5.3.1, § 6.5.2.1	§ 6.5.2.1	REBEL Ch. 4, § 6.5.2.2, § 6.5.2.3
Value Function Approximation	regression architecture, features, basis functions	Ch. 5, § 5.3.1, § 5.3.2	§ 5.3.1	§ 5.3.1	§ 5.3.1	§ 5.3.2
Policy Search	policies, behaviors, hierarchies, MDP decompositions	Ch. 5, § 5.5	§ 5.5.3	§ 5.5.3		EARL GREY § 5.4, § 5.5.2, § 5.5.1
Hierarchical Decomposition	policies, behaviors, hierarchies, MDP decompositions	Ch. 7	§ 7.3.3	§ 7.3.3	§ 7.3.3	

Table 3.1: Approximation types versus PIAGET levels. For each of the approximation types there are four types of learning algorithms. We highlight the main structural and parameter components and for each of the type-PIAGET combinations we refer to the chapters and sections in which relational or first-order versions of these topics are described. We only mention the types of structures; the parameters learned in all abstraction types varies from V, Q, T and R to actions and internal parameters of structures.

Part II

Sequential Decisions in the First-Order Setting

CHAPTER 4

Reasoning, Learning and Acting in Worlds with Objects

Most – if not all – interesting domains are made up of objects and relations. In this chapter we upgrade the algorithms from the previous chapter to relational domains. We start by describing why relational worlds are interesting and which problems they pose to traditional RL algorithms and representations. We find out that in order to lift them to the relational case, we do not have to change the PIAGET principle introduced in the previous chapter. In addition, all of the algorithms introduced in five core abstraction types, can be reused for the relational setting. In order to do this, we need three tools. First, we describe logical representation formalisms based on first-order logic. Second, we describe how to learn first-order, logical abstractions using inductive logic programming and statistical relational learning. Third, we describe how to model actions in first-order logic. Based on these three components we can formalize logically represented, first-order Markov decision processes. It turns out that, using first-order representation, learning and action formalisms, the complete abstraction framework introduced in the previous chapter can be lifted to the relational case. At the end of this chapter, we highlight different views on the new field of relational RL and we discuss what is new in relational RL (for example, the concept of RMDP families) and what is not.

WHEN HUMANS INTERACT WITH THEIR ENVIRONMENT, they usually think of this environment as being composed of *objects*, such as tables, cars, books and houses. These objects are often described in terms of *properties* they have and *relationships* they have with other objects and other people. Representing a world as a set of objects and relations is useful for describing what an agent perceives, but also for describing its goals, intentions and effects of actions (see, the *intentional stance*, Dennett, 1987). If an agent knows how to handle a certain object in the right way, it is probable that this knowledge also applies to 'similar' objects, having similar properties and similar relationships to other similar objects. For example, if I know how to write a note with a pencil, I could expect that I can transfer this knowledge to writing a note with a ballpoint.

Despite theoretical results for classical and abstracted MDPs and practical successes of various abstraction mechanisms for MDPs, propositional techniques do not scale up to larger, and more complex, domains consisting of objects and relations. Nevertheless, most – if not all – complex, real world domains are most naturally represented in terms

of objects and relations. In fact, *“it is hard to imagine a truly intelligent agent that does not conceive of the world in terms of objects and their properties and relations to other objects”* (Kaelbling *et al.*, 2001). The choice for a representation affects which information can be represented, and which kinds of generalization and abstraction can be employed to exploit certain *patterns* and *structure* in a problem. The KR in the previous chapter is based on at most propositional logic, in which the problem is described using a set of features. In this chapter we bring the MDP framework into the relational setting, which involves combining many things, such as first-order logic, probability, utility, and active learning.

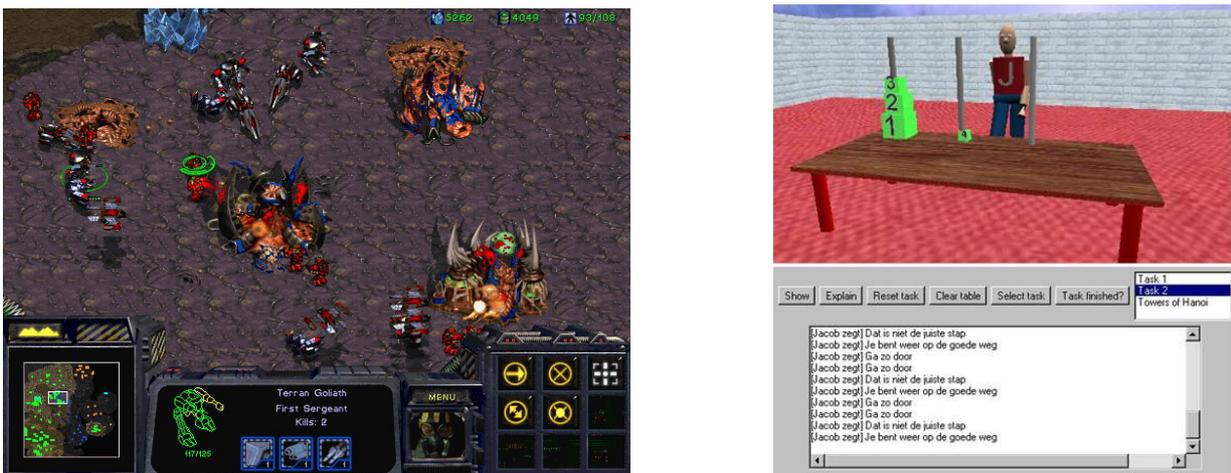


Figure 4.1: a) The real-time strategy game STARCRAFT. b) A 3D virtual world.

Figure 4.1 shows two typical examples of worlds that are best described in terms of objects. The left figure shows a screenshot of the STARCRAFT computer game (by BLIZZARD entertainment), which is an example of a *real-time strategy* game. The goal is to build barracks, factories, and research facilities to construct an army consisting of soldiers and tanks. This army is used to fight against opponents. The right figure shows a typical example of a 3D virtual world in which agents can move around and perform tasks. Both domains show a variety of objects and possible relations between objects and many possibilities for object-based generalization, learning and reasoning. Representing these worlds in terms of propositions is cumbersome, not elaboration-tolerant, and provides no means for generalization over objects.

Logic-Based Artificial Intelligence. First-order logic-based approaches are an essential element in KR issues (Markman, 1999; Sowa, 1999) of many current AI subfields (see Reiter, 2001; Görtz *et al.*, 2003; Russell and Norvig, 2003; Luger, 2002; Brachman and Levesque, 2004; Mueller, 2006). Frege (1879) laid out the foundations for modern first-order logic, though the origins of the idea of having a *universal* language to express knowledge and reason about this knowledge can be traced back to Leibniz. Quoting Davis (2000, p. 4)¹: *“In Leibniz’s vision, something similar could be done for the whole scope of human knowledge. He dreamt of an encyclopedic compilation, of a universal artificial mathematical language in which each facet of knowledge could be expressed, of calculational rules which*

¹This book contains an interesting tour through the development of logic and ideas about computation from Leibniz to Turing.

would reveal all the logical interrelationships among these propositions. Finally, he dreamed of machines capable of carrying out calculations, freeing the mind for creative thought.”

Logic-based AI (LBAI) (Minker, 2000b,a; McCarthy, 2000) has its origins in the seminal work by McCarthy (1959) and the foundations of *situation calculus* (McCarthy, 1963; McCarthy and Hayes, 1969; Lifschitz, 1990). LBAI can be found in such diverse fields as *multi-agent systems* (Fagin *et al.*, 1996; Weiss, 1999; Ferber, 1999; Wooldridge, 2002), ML (see Section 4.3), planning and acting (see Section 4.4), logic programming (see Section 4.2.2.1 and Sterling and Shapiro, 1994; Nilsson and Maluszinski, 1995; Bratko, 2001) and evolutionary computation (Divina, 2006). All of these topics are primarily *applied* logic, and focus on *computational* aspects, combining foundational research with applications. In this chapter we will focus mainly on computational approaches. For coverage of a broader area and history of approaches, see (Minker, 2000b).

Advances in First-Order Logical Machine Learning. Narrowing down from LBAI to ML and datamining, we see that propositional approaches have dominated the literature in recent decades (Mitchell, 1997; Alpaydin, 2004). Popular methods are decision trees, neural networks, support-vector machines, kernel methods (Schölkopf and Smola, 2002) and many kinds of statistical approaches (e.g. mixtures of Gaussian distributions, see Hastie *et al.*, 2001). Usually inputs for the learner are represented as vectors of feature values and generalization employs superpositions of hyperplanes in the input space.

An important limitation of propositional approaches is that they have difficulties with generalization over inputs that have *relational regularities* (Clark and Thornton, 1997; Thornton, 2000). The *parity problem*² is a well-known, extreme case of a problem where the target function to be learned depends on the relationship between (all) inputs, and where clustering approaches (and in fact most propositional generalization techniques, that rely on statistical regularities) are not appropriate (Thornton, 1996). A second limitation of propositional ML approaches is that they cannot deal with *explicit* relational representations of input and output spaces.

Until to a couple of years ago, nearly no effective techniques existed for robust learning and reasoning about objects in probabilistic domains (such as MDPs). In recent years many KR formalisms have been developed for dynamic, object-based, probabilistic domains. Logical ML algorithms have existed for decades (e.g. inductive logic programming; see Section 4.3.2), although limited to supervised learning in static, deterministic settings. To apply logical methods in RL – the main topic of this book – one needs at least mechanisms to deal with uncertainty, concept drift, active learning and decision-theoretic notions. In recent years many combinations of first-order logic, probability and inductive methods have been studied in the ML community (see Section 4.3.3). In addition to the supervised setting, unsupervised learning has been studied in the relational setting in clustering and distance-based settings (e.g. see Ramon, 2002). The supervised and unsupervised setting are concerned with *probabilistic* extensions to logical ML algorithms, or equivalently, with *relational* upgrades of propositional ML algorithms. Relational RL methods can be seen as extending these settings further, by adding *active learning* and *utility*.

²The parity problem is defined as follows. Let $f = \{0, 1\}^n$ be a binary input vector of length n . The parity of f is 0 if the number of 1s in f is an even number (note that 0 is even) and 1 if the number of 1s in f is an uneven number. The parity of a binary vector is dependent on *all* bits.

Pros and Cons of Logic-Based Approaches. Upgrading RL and the MDP framework to work with first-order logic enables much more powerful KR and with that, scaling up to larger and more realistic domains. Logic has, in general, a number of desirable properties (Omar, 1994). First of all, its strong foundations enable transfer of many results concerning e.g. *model theory*, *complexity* and *proof systems*. Second, first-order logic is expressive and incorporates means for many types of *commonsense reasoning* (Mueller, 2006). Third, a very important aspect is that logic enables a *declarative* view of knowledge, i.e. meaning can be independent of the context in which it is used. Knowledge can be stated 'as it is', without specifying it as procedures for use, as independent pieces, facts, or rules. This also forms a basis for *modularity* of knowledge systems, which facilitates a view on (relational) RL as a learning component of an intelligent system, delivering pieces of knowledge that can be incorporated into *integrated* cognitive systems (see the end of this Chapter, and further Chapter 7).

Despite these advantages, debates on the usefulness of (first-order) logic in general AI have been numerous. Discussions on the more general topic of symbolic (to which logical methods belong) versus sub-symbolic (e.g. connectionist systems) go into detail on topics such as *systematicity*, *productivity* and *symbol grounding* (Harnad, 1990) and are beyond the scope of this book (but see Fodor and Pylyshyn, 1988, on this topic). Also, discussions on whether we need explicit representations (e.g. see Brooks, 1991), whether intelligence and representations should be *embodied* (Clark, 1997; Pfeifer and Scheier, 1999) whether (and how) the mind represents concepts (e.g. see Margolis, 1999; Gärdenfors, 2000; Claplin, 2002) and which other types of representations exist for general cognitive systems (see Markman, 1999; Sowa, 1999), are not discussed here. Some other arguments against logic are about the inefficiency of declarative approaches (compared to procedural approaches and *compiled* knowledge, see Omar, 1994), and we will return to this point in this chapter. However, our starting point is the MDP setting as described in the previous chapters and we are interested in lifting that setting to the first-order case, focusing on computational methods.

Still, objections against logic exist in this restricted setting. Well-known problems in logical methods are e.g. *non-monotonic* reasoning, the *frame problem* for dynamic settings, and most importantly, *complexity* of first-order logical approaches. Efficient propositional approaches for reasoning and planning exist, and an important question is always whether we need the full power of first-order logic. We will see that downgrading expressivity will often benefit computational properties.

Lifting the MDP Framework. Introducing relational representations in RL is yet another step in a series of extensions of algorithms and representations for state-transition systems. Figure 4.2 depicts three³ dimensions along which these steps are made. The first dimension ranges from deterministic to stochastic transition systems. The second dimension ranges from discrete representations of states, via propositional to relational representations. The third direction ranges from systems without to systems with program constraints (e.g. task structures and hierarchies). The framework described in Chapter 2 resides at the floor level of the cube, focusing on stochastic systems with no program con-

³One might also introduce another dimension, ranging from discrete to continuous domains. This is beyond the scope of this book, mainly because in the relational setting we focus on discrete problem states. Still there are some advances in enhancing state representations with continuous attributes in the relational setting and we discuss them in Chapter 6.

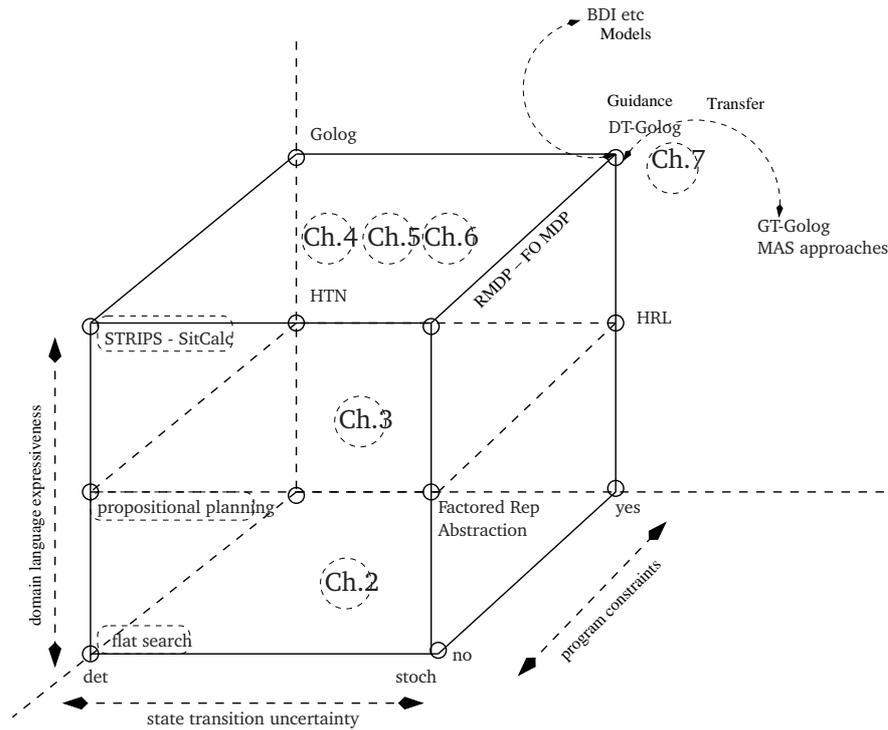


Figure 4.2: *The main dimensions in RL and stochastic planning: 1) a representational dimension, 2) a stochasticity dimension, and 3) a policy bias dimension.*

straints. In Chapter 3 we have discussed the middle level of the cube; systems in which the representation is propositional and transitions are stochastic. Starting from the current chapter, we have arrived at the top level of the cube, in which first-order representations are used. The representational dimension is central in this book, introducing increasingly richer representations in RL. In Chapters 5 and 6 we discuss solution techniques for relational MDPs without program constraints, whereas in Chapter 7 we will discuss models and algorithms that involve program constraints. There are at least two principled reasons for moving towards richer representations for the MDP framework:

1. **Most Interesting Domains consist of Objects.** The domains in Figure 4.1 are only two examples of domains that are most naturally represented in terms of objects and relations, but many other ones can be found throughout the computer science literature. Planning domains such as BLOCKS WORLD and logistics, board games such as CHESS and GO, multi-agent systems and many others are best conceived as domains consisting of a rich structure of objects. Real-world examples include *relational databases, formal ontologies, bio-informatics problems such as mutagenicity prediction, protein secondary structure prediction and three-dimensional structure prediction of proteins, web-and text-mining applications, natural language processing, and citation analysis* (see more in Džeroski, 2001). *Link mining and collective classification* (see also De Raedt and Kersting, 2003) are new methods centered around object-based representations of data. Most of these applications involve representing the data as *graphs*, i.e. as a relational representation over objects, and cannot be handled by traditional, propositional ML algorithms appropriately.

In addition, representing the world in terms of objects and relations is also the most intuitive level to describe goals, mental states, intentions and beliefs of agents. This view is called *intentional stance* (Dennett, 1987) and it is useful to reason about and to predict the behavior of a rational agent acting in the world.

2. **RL should be embedded in Cognitive Architectures.** A second reason for relational representations in RL is that they help to *integrate* RL into more general, cognitive architectures. Logical languages enable *declarative* representation of knowledge, and because of that, possible reuse and transfer of learned knowledge for other tasks. Most general cognitive architectures for commonsense reasoning use a logical representation level for knowledge, beliefs, intentions, goals and plans. If we want to use RL not only as a way of learning an isolated task, but as a mechanism for learning *skills* or behaviors of a general agent, we should be able to use a *unified* language which usually is some form of first-order logic (e.g. see Wooldridge, 2002; Mueller, 2006). This language would then be the core representation of all knowledge and beliefs of the agent and the agent is able to learn and reason in all the same formalism, thereby connecting learning and reasoning (see Dietterich, 2003, for an interesting discussion in a general setting). We will return to this aspect in more detail in Chapter 7. Note that we do not propagate to use first-order logic only; for generally intelligent architectures a multitude of representations will be necessary.

Aims and Outline of this Chapter. This chapter contains two main points. First of all, we stress that once the representation, generalization and action formalisms are upgraded to the relational domains, one can reuse a large majority of the solution techniques we have described in Chapter 3. Throughout this book we will see many examples of this, and towards the end of the chapter we will briefly outline a rough methodology. The message is that the majority of relational RL work consists of "*lifted*" versions of existing (propositional) RL algorithms. There are, however a number of new, challenging problems in relational RL problems and we will describe them in this book, starting from this chapter. The second main point is about efficiency. Throughout this chapter we will describe the core components of various formalisms, though we will bias our description towards restricted fragments of formalisms that exhibit good computational properties. This is because RL and DP algorithms are computationally expensive and the relational context is more complex than the propositional. Keeping the new opportunities of relational representations in RL while, at the same time, keeping the representations and algorithms as simple and as efficient as possible, is vital for obtaining acceptable learning performance. We will see many examples of compact representations, efficient inference algorithms and both representational approximations as well as inferential and sampling approximations.

Upgrading the MDP framework to relational domains involves the following steps. First, we will discuss what a relational representation is and why they are necessary, both from a conceptual or application point of view as well as a technical point of view (Section 4.1). In Section 4.1.3.2 we will define MDPs over relational representations. We will use the BLOCKS WORLD as a running example (Section 4.1.2). Then, following the previous chapter, we need at least three things. First, we need representational formalisms that can talk about objects and relations. For this aspect, we will introduce the formal framework of *first-order logic* in Section 4.2, which will provide a complete KR framework for representing the world, reasoning and compact abstractions. Second, we need ways

to induce abstraction levels over relational domains from data. We will describe the core concepts of *inductive logic programming* and *statistical relational learning* in Section 4.3, providing a framework for learning and generalization over relational domains. Third, we need *action formalisms* that can be used to specify how a relational world changes as an effect of actions and other events. We will describe the key concepts and challenges in using first-order logic in dynamic worlds in Section 4.4. Using these three formal tools, we can define the notion of a *logically represented first-order* MDP in Section 4.5.1, and various forms of state spaces, value functions, policies and reward functions in relational domains. It turns out that the PIAGET mechanism for RL in the face of abstractions still applies once we have upgraded all elements of the MDP solution framework to relational domains. A new element, however, is that we have to deal with *families* of relational MDPs and infinite state and action spaces. Furthermore, relational representations in the MDP framework demand much more attention to be paid for the KR aspects of learning systems; compact representations and efficient inference algorithms are vital in the process of working with first-order logical abstractions. In Section 4.5.2 we discuss relational RL following the PIAGET structure and the five abstraction types defined in the previous chapter. Section 4.5.4 describes different viewpoints on relational RL and Section 4.6 ends the chapter, introducing the material in the rest of the book.

4.1. The World Consists of Objects

For any interesting RL problem some form of abstraction needs to be applied, but in contrast to the previous chapter where we dealt with propositional representations, we want robust learning and reasoning algorithms for *first-order* domains. Many – if not most – interesting (probabilistic) planning domains and decision problems can most naturally be viewed in terms of *objects* and *relations* between objects. First-order representations enable the exploitation of the existence of domain objects, of relations (or, properties) over these objects, and enables the use of *quantification* over objectives (goals), action effects and – most importantly in the context of MDPs – properties of states. Objects are everywhere and we need richer representations and more complex mental states of agents.

Several authors have described the need for, and the possibilities of, first-order formalizations of MDPs early on (Kaelbling *et al.*, 2001; Boutilier, 2001; van Otterlo, 2002). This chapter will cover all aspects of formalizations in line with these works, augmented with concrete descriptions based on work developed in recent years. The level at which we want to deal with objects and relations is that of *symbolic*, logical representations in simulated tasks. Indeed, a complete approach for dealing with objects and relations would mean dealing with a number of difficult aspects ranging from computer vision and symbol grounding to philosophy, metaphysics, cognitive psychology and more. We limit our discussion to concrete algorithmic and representational machinery and give pointers to the literature for a broader view.

4.1.1 Objects are Omnipresent and Indispensable

Humans speak and think about the world as being made up of *objects* and *relations* among objects. There are books, tables and houses, tables *inside* houses, books *on top of* tables, and so on. (Baum, 2004, p. 173): *"We are equipped with an inductive bias, a predisposition to learn to divide the world up into objects, to study the interaction of those objects, and to ap-*

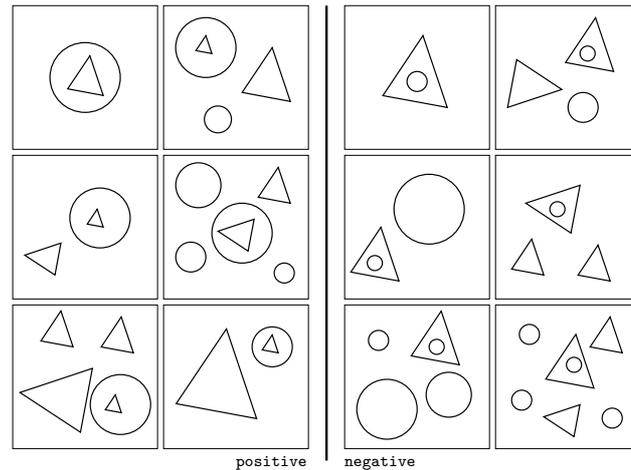


Figure 4.3: A *Bongard problem* (Bongard, 1970). The task in this problem is build a concept of the left example pictures such that no picture from the right side is covered. Propositional representations cannot easily grasp the inherent relational structure.

ply a variety of computational modules to the representation of these objects.” It is our aim to endow computational agents with the same capabilities, because (Kaelbling *et al.*, 2001): “. . . it is hard to imagine a truly intelligent agent that does not conceive of the world in terms of objects and their properties and relations to other objects”. Coherent – either physical or mental – material tends to be well described as an *aggregate* (see Section 3.2.2). (Baum, 2004, p. 168): “The description of the world in terms of objects and simple interactions is an enormously compressed description.”. Such descriptions offer an ability to express generalized information in a compact way that cannot be done without the use of objects.

The main argument for moving towards relational representations is the fact that richer representations are needed for many real-world problems. In the introduction to this chapter we have mentioned several domains where relational representations are useful, such as in *bio-informatics* (Orengo *et al.*, 2003). Most domains such as these are considered in supervised and unsupervised learning research but many similar applications can be found that require an RL approach (e.g. see Figure 4.1). van Laer (2002) provides a good description of the motivations for moving from attribute-value (AV) to relational representations in ML, a view that is shared by most work in logical ML (see Section 4.3.2) and probabilistic versions thereof (see Section 4.3.3) but also by work in KR (see Section 4.2) and action languages (see Section 4.4).

Let us consider BONGARD problems⁴ (Bongard, 1970), a well-known domain that shows clearly why relational representations are useful. Figure 4.3 shows a typical example. The task – in a supervised learning setting – is to obtain a classifier that will assign class value *positive* to all six examples on the left and *negative* to all six examples on the right. Each example consists of a number of geometric⁵ figures scattered around each example. A classification rule for the positive examples could be “if there is a triangle inside some circle then class is positive” and for the negative examples “if there is a circle inside some triangle then class is negative”. Remember that an attribute-value representation consists of a vector $\langle f_1 = r_1, \dots, f_n = r_n \rangle$ assigning a value r_i to each *feature* f_i ($i = 1 \dots n$) in

⁴See for descriptions (Hofstadter, 1979, Chap. XIX) and (van Laer, 2002, pp. 46–47).

⁵BONGARD problems are not limited to this kind of geometric figures. There are many others, containing lines, dots, patterns and many more figures (see Bongard, 1970).

the domain. The BONGARD examples consist of a variable number of objects (e.g. circles, squares, triangles, etc.) with a varying number of properties (e.g. color, shape, size, orientation, etc.). When we look at an AV representation of these BONGARD problems, we can notice several difficulties (De Raedt, 1998; van Laer and De Raedt, 2001a):

- One should fix the maximum number of objects in an example. Given a bound b on this number, one can then list attributes f_i^1, \dots, f_i^n for each object i ($i = 1 \dots b$). Many of these attributes will have a `nil` value, since not all examples will contain b objects and not all objects will have all attributes specified.
- One should *order* the objects in an example scene. This is a problematic task for example in the BONGARD problems in Figure 4.3, where all objects are scattered around the scene. There are exponentially many orderings (in the number of objects) and abstractions and generalization will also depend on this order.
- Each possible relation between objects should be represented as a separate attribute, and these attributes should be ordered. This number grows exponentially in the number of objects and again, the order is problematic.

BONGARD problems, as well as the STARCRAFT example in Figure 4.1 and the BLOCKS WORLD we will discuss later, are obviously toy problems. Still, they are very similar to many real-world problems where the same KR issues arise. In many domains the maximum number of objects is not known beforehand. Fixing the numbers of objects, attributes and relations, as well as ordering them, is often problematic and unnatural. Even if one does manage to represent a problem such as the BONGARD problem using an AV representation, there is still the impossibility of compactly abstracting and generalizing over many *structural patterns* in the problem. A classification rule for the positive class in Figure 4.3 that makes use of *a triangle inside some circle* would require a large disjunction over many attributes. Let `insideTCij` stand for the attribute denoting whether triangle i is inside circle j , then the classifier contains the rule `insideTC11 \vee ... \vee insideTC m n` , where m and n are the maximum numbers of triangles and circles. In Section 4.2 we will see how to represent compact relational statements using first-order logic using *quantification over objects*. For more on the relations between propositional and relational representations with an emphasis on ML, see (De Raedt, 1997, 1998; van Laer and De Raedt, 2001a).

A second class of arguments in favor of relational representations is related to a more general view on KR; to see RL as a *component* in more general AI architectures (van Otterlo, 2002; van Otterlo *et al.*, 2003). Because our aim is to use relational representations in RL, and because there are many connections with other fields such as *intelligent agents*, *planning* and *cognitive architectures*, it is natural to look into the representational standards in these fields. Relational representations, and more specifically first-order logic, allows for compact, expressive and elaboration-tolerant specification of general knowledge. The use of knowledge in the learning process is best facilitated by a first-order logical language. Knowledge can be used in the form of a domain theory such as in action theories (see Section 4.4). Knowledge can also be used to *guide* learning, by providing an initial theory or policy, or it can be used to help the construction of hypotheses (such as in first-order induction, see Section 4.3). A first-order logical language is also useful for *transfer* of learned knowledge to other, related domains. Several relational RL systems have shown

that learned policies can be easily translated to new problems (e.g. see Mellor, 2005b; Driessens *et al.*, 2006a; Stracuzzi and Asgharbeygi, 2006). First-order *goal specifications* allow learning about multiple concrete goals that differ only in which specific objects play a role in the goal. For propositional representations, transfer is possible to some extent (e.g. see Asadi *et al.*, 2006, for an HRL example) but still limited compared to the relational setting. Embedding RL into more general AI architectures is the topic of Chapter 7.

In summary, the use of first-order representations is **i)** needed for representing and generalizing over objects, properties and relations, **ii)** required for a proper generalization of AV representations that fixes some problematic aspects of representing object-based domains propositionally, **iii)** essential to embed RL into existing architectures in intelligent agents, planning and cognitive architectures, and **iv)** useful for using and transferring knowledge in (reinforcement) learning processes.

Objects and Relations in RL. Relational representations in RL offer many new possibilities, such as learning in worlds with objects, generalization over concrete objects and abstract goals. For example, a general task in the BLOCKS WORLD might be to *build two towers using at least 7 blocks*. This task has many states that satisfy this goal, and more importantly, it generalizes over concrete blocks and a learned policy can be independent of the total number of blocks (as long as it is more than 6).

The goal of learning is a relational *policy* that compactly represents how to generate actions, using relational properties of states and actions. For this, one can learn relational *value functions* or directly learn policies (e.g. policy search, see Section 3.7). The advantage of relational formalisms is that domain knowledge can be easily incorporated in the form of first-order logical knowledge bases. Model-free relational RL methods typically require *numerical regression* algorithms that can handle relational representations, whereas model-based algorithms mainly employ logical *deduction* using the domain specification. Both settings are described in Chapters 5 and 6 respectively. For more general information about the use of RL in domains made up of objects and relations, we refer to (Boutillier, 2001; Kaelbling *et al.*, 2001; van Otterlo, 2002; Džeroski, 2002; van Laer, 2002; van Otterlo and Kersting, 2004; Tadepalli *et al.*, 2004; van Otterlo, 2005).

Related Areas. Relational representations are centered around objects and relations. In this book we focus mainly on logical languages based on FOL, though the computer science literature contains many other, similar or related KR schemes. Among these, relational database systems, and *object-oriented* (OO) modeling and programming languages are widespread. Brachman and Levesque (2004, Chap. 8) discuss OO models with *inheritance hierarchies* and classes, and in addition *frame-based systems* (see Minsky, 1985). These OO models and frames differ in the way information is structured. Whereas in a set of FOL sentences information can be scattered around through the database, information in these models is centered around the objects and classes. Related formal systems are *structured descriptions* and *description logics* (Brachman and Levesque, 2004, Chap 9, but see also Section 4.2.2.3). For many more similar representations, see (Markman, 1999; Sowa, 1999; Russell and Norvig, 2003; Brachman and Levesque, 2004).

Outside computer science, for example in philosophy and cognitive or developmental psychology, much research has been devoted to questions about what objects are, how humans see them, whether they actually exist and so on. Especially relevant are questions about how humans acquire the object concept, and when. These questions are beyond the

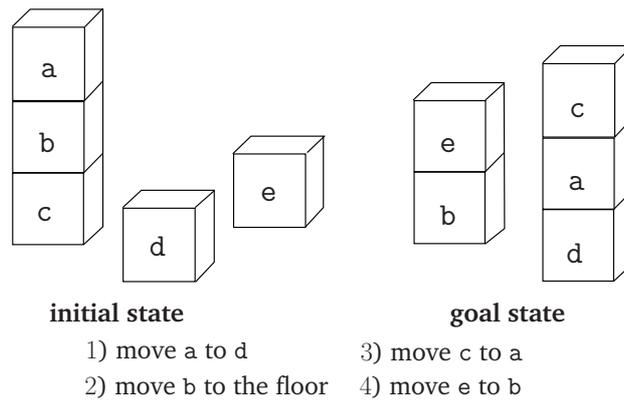


Figure 4.4: A BLOCKS WORLD planning problem and an optimal plan.

scope of this book, but see (Piaget, 1950; Margolis, 1999; Gärdenfors, 2000; Baum, 2004) for interesting ideas and pointers and (Baillargeon, 1999) for interesting experiments in developmental psychology.

4.1.2 A Relational Domain: BLOCKS WORLD

Out of many artificial domains, the BLOCKS WORLD (Slaney and Thiébaux, 2001) is probably the most well-known problem in areas such as KR and planning (see Russell and Norvig, 2003; Schmid, 2001; Brachman and Levesque, 2004) and recently, relational RL. In itself, it may not be of much practical interest. However, it is a hard⁶ problem for general purpose AI systems, it has a relatively simple form, and it supports meaningful, systematic and affordable experiments. Because the methods in this book extend classical decision-theoretic planning methods to relational domains, it is natural to consider a domain that has had so much attention in the (deterministic) planning community. Indeed, the large majority of relational RL systems described in this book uses BLOCKS WORLD as their main experimental domain. This includes the new methods we describe in this book (see Chapters 5 and 6). The BLOCKS WORLD can be seen as a *Drosophila*⁷ for planning tasks and relational RL; it is one of the simplest cases that still possesses much of the structure and complexity of many other, possibly more complicated, domains (see also Baum, 2004, par. 8.2 for an interesting discussion). Throughout this chapter we will use BLOCKS WORLD as our running example to explain various concepts in representation, learning and acting.

The standard BLOCKS WORLD formulation consists of a number of cubic-shaped *blocks* stacked into towers on a floor large enough to hold them all, see Figure 4.4. Each configuration of blocks is a *state*, and an agent is provided with deterministic *operators* (i.e. actions) that can be used to move blocks either on top of each other, or onto the floor. Sometimes a separate *grripper* is modeled that can *hold* a block and pick up or put down a block (e.g. see Pasula *et al.*, 2004, that also provides a physical simulation). A BLOCKS WORLD *planning problem* is defined by an *initial state* and a *goal state*. A *solution* to the planning problem is a *plan* (i.e. sequence of operator applications) that will transform the

⁶This is a returning pattern in AI; problems that are difficult for humans (e.g. CHESS, huge calculations) are often relatively feasible for computers whereas problems that seem so easy for humans (visually recognizing objects, playing with blocks, natural language) prove very often difficult for computers.

⁷The *Drosophila* is a family of fruit flies that has enabled biologists to do cheap experiments in genetics.

k/n	0	1	2	3	4	5	6	7	8	9
0	1	1	3	13	73	501	4051	37633	394353	4596553
5	1	6	43	358	3393	36046	424051	5470158	76751233	1163391958

Table 4.1: *Some blocks world sizes for k grounded and n ungrounded towers.*

initial state into the goal state (see also Section 2.4). Note that most systems discard most of the physics; the BLOCKS WORLD is used as an entirely artificial, symbolic problem.

Initially there were three theoretical problems that prevented BLOCKS WORLD to be an efficient benchmark problem: a lack of knowledge on how to generate suitable problem instances for systematic experimentation, a lack of knowledge about complexity classes and performance time and solution quality for various BLOCKS WORLD-specific planning methods, and an ignorance of BLOCKS WORLD-specific knowledge that general systems should be able to exploit in order to do well in this domain. The lack of theory was gradually solved in the recent decades and much is known about the complexity of representing, learning, planning and reasoning in BLOCKS WORLD problems (Slaney and Thiébaux, 1994, 2001).

Let us first consider the *size* of problems, which grows fast in the BLOCKS WORLD. Let k be the number of grounded⁸ and n the number of ungrounded towers. The total number of distinct states can be computed using:

$$g(n, k) = \sum_{i=0}^n \binom{n}{i} \frac{(n+k-1)!}{(i+k-1)!} \quad (4.1)$$

An equivalent inductive definition shows more clearly the relationship between different world sizes. One starts with base case $g(0, k) = 1$ and further computes using $g(n+1, k) = g(n, k+1) + (n+k)g(n, k)$. Figure 4.1 shows sizes for blocks worlds up to 9 blocks. For 10 blocks the number of states (with only ungrounded towers) is more than 59 million, and for 30 blocks it is approximately 1.98×10^{35} .

A further question – given the combinatorial character of the problem – is how to *generate random instances* of BLOCKS WORLD problems. This is an important aspect for the validation of planners, or training of learning systems. It turns out that generating random BLOCKS WORLD states is a non-trivial task. Equation 4.1 can be used to compute the underlying distribution and Slaney (1995) provides an efficient algorithm. In Chapter 5 we elaborate on this when discussing model-free relational RL algorithms.

Deterministic BLOCKS WORLD problems have been used widely in the planning community and much is known about the complexity of various planning tasks (Gupta and Nau, 1992; Helmert, 2003). Much depends on whether optimal or non-optimal plans are required, whether goals are completely specified and various other characteristics of the domain and the specific task. Indeed, many different versions of BLOCKS WORLD, but also different representations (e.g. see *deictic* representations further in this section), are used throughout the literature (e.g. see Whitehead and Ballard, 1991; McCallum, 1995; Finney *et al.*, 2002b,a). In this book, we focus mainly on BLOCKS WORLD problems specified as MDPs (or, more specifically, as *relational* MDPs) where the environment behaves probabilistically and where the goal is to find a *policy* (or a *universal plan*) that is optimal with

⁸Grounded means that the tower ends on the floor.

respect to a reward function definition and performance criterion.

Other Domains. BLOCKS WORLD is often used because it is relatively simple (to explain and to model) while still being quite complex for general planning and ML algorithms and it easily enables various problem sizes. Furthermore, it can be extended with uncertainty and more complex events. For example, one can add uncertainty in state transitions or in observations (e.g. as in POMDPs), but one might also consider more severe extensions such as physically real simulations with slippery or exploding blocks and so on.

There are several other types of problems that are typically used in planning and relational RL systems that share characteristics with the BLOCKS WORLD. In the field of relational RL we will see computer games such as TETRIS, DIGGER, TIC-TAC-TOE and GO, but also various forms of *logistics domains* (see Chapter 6) where the task is to move certain objects such as *boxes* to various places. The *international planning competition* (IPC) was recently extended with a probabilistic track (Younes *et al.*, 2005) and many of these IPC problems provide interesting test cases for relational RL systems (e.g. see Fern *et al.*, 2006). Interesting examples that are more complex and still only tractable if simplified much, are *real-time strategy games* (e.g. see Guestrin *et al.*, 2003a, and Figure 4.1), *adventure games* (Amir and Doyle, 2002) and *real-time shooter games* (e.g. see Jacobs *et al.*, 2005, for an initial approach). The BLOCKS WORLD maintains its position as a main test bed for current state-of-the-art relational RL though many interesting domains await.

4.1.3 Representing a World of Objects and Relations

Representing the world in terms of objects and relations requires representational devices beyond the level of simple propositions. More specifically, for relational RL – a computationally demanding task – we want representations that can be learned and used efficiently. In this section we define *relational representations* in *extensional form*, i.e. a symbolic language to characterize concrete states. In Section 4.2 we extend our language to incorporate means for general statements and reasoning mechanisms *about* these concrete states. We focus on symbolic, logical formalizations, but see e.g. (Margolis, 1999; Gärdenfors, 2000; Sowa, 1999) for other directions.

4.1.3.1 RELATIONAL REPRESENTATIONS

The problems of AV representations we have described in Section 4.1.1 can be overcome by looking at *relational* (or *first-order*) representations. Such a relation presupposes a *language* to express *relational facts*. A *relational alphabet* Σ consists of a set of relation symbols P and a set of constants C . Each constant $c \in C$ denotes an object in the domain and each $p/a \in P$ denotes either a property (or attribute) of some object (if $a = 1$) or a relation between objects (if $a > 1$). For the BLOCKS WORLD in Figure 4.5a, we see that we have 6 constants, which are the 5 blocks $\{a, b, c, d, e\}$ and the floor. Blocks can be on top of something, and they can be clear, which is specified using the predicates $on/2$ and $clear/1$. In the example, the *fact* $on(a, b)$ is true, and $clear(c)$ is not. Blocks might have a *color* and for this we would need⁹ new predicates such as $yellow/1$ and $blue/1$

⁹Note that properties of objects might also be defined in terms of binary predicates and the addition of new constants that denote the properties. In the example, one might introduce the new constant *yellow* and a new predicate $color/2$ such that $color(a, yellow)$ denotes that block *a* has color *yellow*. Which type of modeling is used, depends on technical and notational convenience.

Relational (or first-order) representations of *examples* are *sets of relational facts*. Each example can be characterized by only those facts that hold in the example. The BLOCKS WORLD in Figure 4.5a can be represented as

$$\{\text{clear}(\text{a}), \text{on}(\text{a}, \text{b}), \text{on}(\text{b}, \text{c}), \text{on}(\text{c}, \text{floor}), \\ \text{clear}(\text{d}), \text{on}(\text{d}, \text{floor}), \text{clear}(\text{e}), \text{on}(\text{e}, \text{floor})\}$$

Note that the number of facts in an example is not fixed, and that the order of the facts is arbitrary. Notice also how compact this description is compared to a propositional version that would consist – among others – of 6×6 on-propositions of which only 5 are true as can be seen in the relational representation. In fact, AV representations are only a special case of relational representations (De Raedt, 1997; van Laer, 2002). De Raedt (1998) shows that for the setting where examples are sets of true facts, complete propositionalization is possible, but results in learning problems that are – in size – exponential in the number of parameters of the original learning problem. A propositional representation of a relational problem throws away much of the inherent structure of the relational facts in the problem representation. For example, the structure of $\text{on}(\text{a}, \text{b})$ shares the parameter object a with the fact $\text{clear}(\text{a})$. This structure can be exploited for compact abstractions, something which is not possible in a propositional representation. In Section 4.2 we will define first-order examples more formally as *first-order structures* and *Herbrand interpretations* of a given first-order language.

The number of possible examples for a given relational alphabet can be huge. Let $P = \{p_1/\alpha_1, \dots, p_n/\alpha_n\}$ be a set of predicates with their *arities* and let $C = \{c_1, \dots, c_k\}$ be a set of constants. For each of the α_i arguments of a predicate p_i ($i = 1 \dots n$) we can choose $|C|$ constants. We can view each ground relational atom as a binary feature, i.e. it either holds in a state or not. Therefore, the number of world states $|S|$ based on P and C is

$$|S| = \prod_{i=1}^n \left(2^{|C|}\right)^{\alpha_i} \quad (4.2)$$

The amount of states grows double exponentially. For most domains however, the number of *legal* system states is much lower than the number that can be generated syntactically from the predicates and constants. For example, with four blocks and a floor and the *on* and *clear* relations one can construct $2^{25} \times 2^5 = (1024)^3$ states. This huge number is due to the fact that facts such as $\text{on}(\text{floor}, \text{floor})$ are included. The actual number of states is only 73 (see Table 4.1 for ungrounded towers). Excluding illegal states can be done by a *background theory* (see Section 4.2).

There is a strong connection between relational representations and standard relational *databases*. Propositional representations correspond to databases consisting of a single table where each example is a row in the table. Relational representations correspond to multiple tables where each relation is a separate table in the database. See more on the relation with databases in (Džeroski and Lavrac, 2001a).

Functors. An additional element in relational representations frequently encountered are *function symbols*, or *functors*, which can be used to represent *structured terms*. Formally, a relational alphabet can be extended with a set $F = \{f_1/\alpha_1, \dots, f_k/\alpha_k\}$ where each f_i ($i = 1 \dots k$) denotes a function from C^k to C , where C is the set of constants. For example, the structured term $\text{mother}(\text{martijn})$ maps to the constant greet . Simple functor occurrences such as $f(\text{a}) = \text{b}$ can be transformed into relational form $r(\text{a}, \text{b})$, i.e. as a transformation

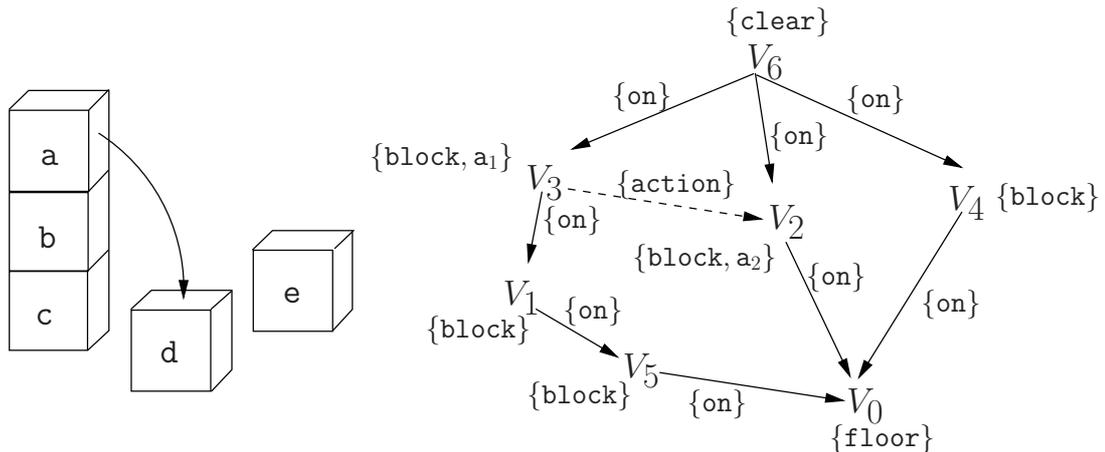


Figure 4.5: a) An example BLOCKS WORLD state and action. b) A *graph representation* of the state-action example in the left figure (Gärtner et al., 2003; Driessens et al., 2006b).

of a functor f/α to $r/(\alpha + 1)$. *Recursively defined* functors are more difficult. For example, the *successor* functor on natural numbers allows the construction of infinite terms such as $\text{succ}(\text{succ}(\dots(\text{succ}(1))))$. Once there are functors such as these, a substained generative process creates an infinite number of facts and this cannot be collapsed into a finite number of simple relations.

Generalization and Abstraction. Relational representations are *symbolic* in nature. On the contrary, AV representations can be mapped onto a real-valued vector such that standard distance metrics (e.g. Euclidean, Manhattan) can be used to compute *similarity* (i.e. distance) between examples for generalization purposes. Generalization for relational representations is usually performed based on first-order versions of *version spaces*. In addition, distances (and kernels) have been defined for relational representations too (see Section 4.2.3). The main challenge is to exploit the inherent structure of relational facts as well as the structure shared among several facts or several examples (using *variables* to range over constants) for generalization and abstraction purposes. These matters will be treated at length in the remainder of this chapter, and more specifically in Section 4.3.

Graph-Based Formalisms. Most of the formalizations we describe in this chapter are based on logical *languages*, but alternatives exist. *Graph-based* representations are popular, visual representations of structured data in e.g. probabilistic reasoning (Jordan, 1999) and factored MDPs (see Section 3.5). For relational data many types exist (see De Raedt and Kersting, 2003, and Section 4.3.3). The use of graphical models in relational RL has, so far, been limited to PRMs (Guestrin et al., 2003a), and *relational attribute graphs* (Gärtner et al., 2003; Driessens et al., 2006b; Dabney and McGovern, 2006, 2007).

An example of a BLOCKS WORLD-specific, relational attribute graph is depicted in Figure 4.5. In the left figure a BLOCKS WORLD state is shown where the arrow denotes the action $\text{move}(a, d)$. The right figure shows a graph representation of this state-action pair. Each block and the floor are represented by vertices, and relations (including $\text{move}(a, d)$) are shown as edges. Notice the special vertex *clear* that is connected to all clear blocks. Driessens et al. (2006b) use this graph representation as input to a *graph kernel regression* algorithm, i.e. the graph representation is used as an intermediate representation

for Q -value estimation. The difference with the representation used by Dabney and McGovern (2006, 2007) is that the latter use the graphs as the base representation for their algorithms.

4.1.3.2 RELATIONAL MARKOV DECISION PROCESSES

Relational representations of states and actions can be used to form *relational* MDPs (RMDP). The majority of relational RL work is based on this model, though most RMDPs are implicitly specified using high-level specifications using a logical language (see more on this in Section 4.5.1.1). Here we introduce a basic relational MDP formalization, assuming we are provided with a language that at least contains predicates, constants and action symbols.

DEFINITION 4.1.1 ▶ Let $P = \{p_1/\alpha_1, \dots, p_n/\alpha_n\}$ be a set of predicates with their arities, $C = \{c_1, \dots, c_k\}$ a set of constants, and let $A' = \{a_1/\alpha_1, \dots, a_m/\alpha_m\}$ be a set of actions with their arities. Let S' be the set of all ground atoms that can be constructed from P and C , and let A be the set of all ground atoms over A' and C .

A **relational Markov decision process (RMDP)** is a tuple $M = \langle S, A, T, R \rangle$, where S is a subset of S' , A is defined as stated, $T :: S \times A \times S \rightarrow [0, 1]$ is a probabilistic transition function and $R :: S \times A \times S \rightarrow \mathbb{R}$ a reward function.

This definition provides the core model used in the remainder of this book. The difference between RMDPs and MDPs (see Definition 2.2.1) is only the definition of S and A , whereas T and R are defined as usual. As we have explained, there are now many possibilities for abstraction due to the structured form of ground atoms in the states. Note that in RMDPs there is an additional opportunity in the structured form of actions. Because states and actions share parts of the structure (e.g. constants) abstraction formalisms can – and must – take advantage of this, and in the remainder of this book we will deal with this aspect extensively. Another opportunity for abstraction are similarities between RMDPs that are defined using different sets of constants. Indeed, a BLOCKS WORLD with five blocks will not differ much conceptually from one that contains six blocks. In Section 4.5.1.1 we will discuss *families* of RMDPs, which are sets of related RMDPs that differ only in the number of constants and possibly the definition of goals (i.e. reward function).

EXAMPLE 4.1.1 ▶ An example of a five-block world based on the predicate set $P = \{\text{on}/2, \text{cl}/1\}$, the constant set $C = \{a, b, c, d, e, \text{floor}\}$ and the action set $A' = \{\text{move}/2\}$ we can define the blocks world containing 5 blocks with $|S| = 501$ legal states. T can be defined such that it complies with the standard dynamics of the BLOCKS WORLD move operator (possibly with some probability of failure) and R can be defined such that it sets a positive reward for states that satisfy some goal criterium and 0 otherwise.

A concrete state s_1 is $\{\text{on}(a, b), \text{on}(b, c), \text{on}(c, d), \text{on}(d, e), \text{on}(e, \text{floor}), \text{cl}(a)\}$ in which all blocks are stacked. The action $\text{move}(a, \text{floor})$ moves the top block a on the floor. The resulting state s_2 would be $\{\text{on}(a, \text{floor}), \text{cl}(a), \text{on}(b, c), \text{on}(c, d), \text{on}(d, e), \text{on}(e, \text{floor}), \text{cl}(b)\}$ unless there is a probability that the action fails, in which case the current state stays s_1 . If the goal would be to stack all blocks, state s_1 would get a positive reward and s_2 0.

Note that for a relational representation, the number of states (and actions) that can be constructed from the predicates and constants is usually much larger than the number

of *legal* states. Definition 4.1.1 is *domain-independent* in the sense that the state-action space is constructed purely from the language elements. For example, the above definition includes states that contain the atom $\text{on}(a, a)$, which is an illegal element of any state in a standard BLOCKS WORLD. A *domain-specific* theory can limit this number by excluding illegal states. For example, an RMDP M based on $\{\text{on}/2, \text{clear}/1\}$, $\{a, b, c, \text{floor}\}$, and $\{\text{move}/2\}$ has only 13 legal states, but the number of possible world states is 2^{20} . Extending the set of constants with one block results in 2^{30} states, but only 73 legal ones. For now, we can assume that an implicit domain theory limits the RMDP to contain only legal states and actions.

4.1.3.3 FROM PROPOSITIONAL TO RELATIONAL

Increasingly richer representations of states and actions range from *discrete* (Chapter 2) to *propositional* (Chapter 3) to *relational* or *first-order* (this chapter). We have argued that for many domains relational representation and generalization is most natural and preferred, and more powerful (De Raedt, 1997, 1998). Still, relational languages and abstraction require more complex systems and the state-of-the-art in propositional abstraction mechanisms for MDPs is much more advanced (see Chapter 3).

Several approaches have investigated the use of representations that are strictly less powerful than relational representations for RMDPs. An interesting recent example is the *object-oriented* representation for RL introduced by Diuk *et al.* (2008). Based on *classes* of objects, attributes and relations, a compact representation can be constructed that is still propositional, and a *provably efficient* algorithm can be devised for learning in (deterministic) environments.

Roughly, one can identify three approaches that use essentially propositional features for model construction, such that classical, propositional ML can be used for learning. The feature construction process is the phase where the approaches differ.

Many-Layered Architectures and Constructive Induction. The first approach are full-propositional representations that simply ignore the relational structure of states and actions. *Naive* propositionalization treats each ground atom as a binary feature, but an additional possibility is to model the problem using a more limited set of hand-coded or automatically constructed features. In principle, it is possible to translate every (finite) relational representation into a binary state vector that can be used as input for any propositional learning algorithm described in Chapter 3. As we have seen in the previous section, this generates a huge state vector and furthermore such representations inherit all problems concerning the naming and orderings of atoms and constants.

Irodova and Sloan (2005) describe experiments on BLOCKS WORLD problems using cleverly engineered, propositional features and linear value function approximation. This yields very fast run-times compared to other relational RL systems. However, this – and the possibility of generalizing to larger domains – is heavily dependent on ad-hoc, domain-dependent feature engineering. Baum (2004, p. 154) did similar experiments using MLP-based generalization on a raw input (i.e. a full-propositional representation) which failed for even very small BLOCKS WORLDS and concluded “[t]hese are very complex concepts, hard to learn or even to represent as a neural net”. In other work, (Baum, 1998, 1999) he reports on very successful experiments using evolutionary policy learning using the more complex language of *S*-expressions for policy representation.

Related approaches exist under the general name of *constructive function approximation*¹⁰ (CFA). Clark and Thornton (1997) (but see also Thornton, 2000) explore the statistical properties of the interactions and dependencies between propositional features. Utgoff and Precup (1998) describe methods that are based on that and which can – if applied to RL problems – be classified as *adaptive resolution* techniques (see Section 3.4.2). CFA methods build complex features from simple ones by exploring the relational interactions between features. The *many-layered learning* method described by Utgoff and Stracuzzi (2002) builds a large number of layers of features, each one building on the previously introduced features, in a card game in which complex relations are needed.

First-Order Features and Propositionalization. A second way to avoid full relational representations and learning algorithms is by *propositionalization*, which is defined by a transformation of an existing relational representation to a propositional one. Kramer *et al.* (2001) provide a recent survey of various approaches that start with a relational representation of examples (and possibly a background theory) and construct *first-order features* from this relational description. A first-order feature describes a number of first-order properties that are of interest, using a logical language (see Section 4.2). For example, in a standard BLOCKS WORLD representation where relations about towers and tower heights have been added as background knowledge, a first-order feature might be induced or constructed that states that “*there are three towers with an average height above 4*”. Such a feature is either true or false and can be used as a propositional feature for model construction. Multiple features can be found by various descriptive induction algorithms such as *association rule mining* or *frequent itemset mining*.

There are several relational RL algorithms that make use of propositionalization. Whereas Sanner uses first-order features to compactly represent first-order value functions either in a naive Bayes decomposition (Sanner, 2006a) or as a linear combination (Sanner and Boutilier, 2006), Walker *et al.* (2004) use first-order features as input of a kernel regression algorithm. The generation of first-order features can be done in various ways. Sanner (2006a) uses splitting and merging of simple features, Sanner and Boutilier (2006) use both hand-coded features and features deduced from a domain theory, and Walker *et al.* (2004) use an off-line feature generation mechanism (see Srinivasan, 1999).

Deictic Representations. *Deictic*¹¹ representation (DR) is an interesting third alternative for relational representations. DR has been used in linguistics and was introduced into AI by Agre and Chapman (1987) and subsequently used in other work in reactive behavior learning (Whitehead and Ballard, 1991; Benson, 1996). A deictic expression “points” to something. Its meaning is relative to the agent that uses it and the context in which it is used (Kaelbling *et al.*, 2001). Examples are “*the book that I am holding in my hand*” and “*the door in front of me*”. DR avoids the difficulties of handling relational representations but do have some capability of generalizing over objects. For example, what will happen with “*the thing in my hand*” if I drop it, is the same for all objects that I can have in my hand: it falls. The main advantage of DR is that it avoids the arbitrary naming of objects. Deixis can be used for state representations, but also for actions. Rather than have an action like `pickup(B)` that picks up a named block, we might have a generic `pickup` action

¹⁰Similar mechanisms in relational learning, where new, first-order features are constructed during learning, are known under the name of *predicate invention*.

¹¹Deictic comes from the Greek word DEIKTIKOS which means “*able to show*”.

that always picks up the block that is currently in focus.

The core of a DR is formed by so-called *markers* which can be placed upon objects and moved around, and of which one is the current *focus*. Information about the current state consists only of properties of marked objects, which renders the state partially observable; in exchange for focusing on only a few things, the agent loses the ability to see the rest. Propositional representations yield large observation spaces but full observability, while DRs yield small observation spaces but partial observability. An advantage of DRs is that the size of the observation space does not grow when additional objects are added. On the other hand the action space is larger than in a full propositional representation. In addition to the available actions that can be performed on marked objects, the action space is enriched with additional actions for moving the markers.

Thus, DRs enable a kind of generalization over object-based domains, but yield partially observability and larger action spaces. Actions have to be learned for the original action space, but in addition for moving the markers too. Together they tend to generate a difficult POMDP, demanding a solution based on histories of actions and observations to behave effectively (see Section 2.7). Finney *et al.* (2002b,a) report on model-free experiments with DRs in BLOCKS WORLD, comparing propositional and deictic representations using both an MLP function approximator (see Section 3.6.2.2) and the *G*-algorithm tree learner (see Section 3.6.2.3). Some history of past observations and actions is included as input in the VFA. Results show that when using MLPs the full propositional representation outperforms the DRs in terms of total reward per trial, but the opposite is true when using the *G*-algorithm. However, when using the *G*-algorithm, none of the representations reaches the performance level of the MLP approach, unless the trees are allowed to grow very large. The main problem for DRs seems to be the exploration problem; an action set that includes the ability for the agent to control its own attentional focus inherently increases the difficulty of the exploration problem by allowing it to easily spend a lot of time exploring a useless part of the state space. The larger the domain is, the more locations a marker can be placed upon and this creates a large exploration space. A similar partially observable BLOCKS WORLD was used by Dabney and McGovern (2006, 2007) in a relational version of the UTREE (see Section 3.6.2.3) algorithm.

4.2. Representation and Inference in First-Order Domains

To compactly represent RMDPs and in addition to reason and learn about them, we will use the formal tool of *first-order logic* (FOL). Logical formalisms such as FOL generally provide the following things: **i)** a *syntax*; a *language* (e.g. keywords, symbols, *formulas*) to represent and reason over the world, **ii)** *semantics*; the kind of (mathematical) structures that correspond to the domain (including its *ontology*) one wants to model, e.g. an RMDP, **iii)** *model theory*; how the syntactic structures relate to the semantic structures (e.g. what it means for a syntactic description to be *true* in a semantic structure), **iv)** *proof theory*; how one can do (commonsense) *reasoning* in the language, and **v)** *meta-logical* aspects; how to do reasoning about the logic itself. The description in this chapter will first describe these aspects in sufficient detail to understand the material in the remainder of this book.

In the remainder of this section, we discuss some variations on FOL that differ in expressiveness of the language, in the structures that can be reasoned about and in the reasoning mechanisms themselves. A little more detail is provided on *Horn logic* and

the *logic programming language* PROLOG because it provides a formal basis for much of the work done in relational RL in general, and the algorithms and systems described in Chapters 5 and 6 in particular. This section lays down a KR basis that we use in Section 4.3 for *inductive* reasoning. In Section 4.4 we extend it to reason about action and change.

Abstraction and generalization over objects is a powerful and desirable property of FOL. However, specifically for ML, and more in general, in relational RL, it comes with a number of challenges. Reasoning in FOL, and manipulating first-order abstraction levels is significantly more computationally demanding than in the propositional setting. One of the main deficiencies in the first-order setting is the lack of a natural *distance measure* (or: *similarity measure*) which is often taken for granted in many propositional ML algorithms. Any propositional feature representation can be interpreted as a vector in the \mathbb{R}^n space where the standard *Euclidean* or *Manhattan* distance measure applies. In the first-order setting, there is no such space and other methods have been developed to provide it. In Section 4.2.3 we discuss first-order *partitions*, *distances* and compact *data structures* whereas in Section 4.3 we formalize an efficient ordering of first-order abstraction levels in a restricted setting.

Other representations exist in relational RL that bear similarity to FOL such as *graph-based* (Dabney and McGovern, 2006) and *object-oriented* (Guestrin *et al.*, 2003a) formalisms, though in this chapter we take a traditional approach based on a *language* and its *semantics*. We do this because of two reasons. First, most methods in logical ML, action formalisms and relational RL are of this kind. Second, it is our conviction that it provides a foundation for understanding other formalisms that share – in principle – the same semantics, i.e. aimed at representing and generalizing over objects and relations.

4.2.1 First-Order Logic

The formal language of FOL has its origin in the work by Frege (1879) and is since the work by McCarthy (1959) (see also Lifschitz, 1990) adopted by the AI community as a main formalism for KR. In recent decades a jungle of languages has appeared, that more or less have their roots in FOL, all providing means to model systems that can best be described in terms of *relational structures*. Pure FOL is a standardized language and it is often put forward that it is powerful enough for most reasoning tasks, but many useful fragments and extensions have been proposed that offer various advantages for particular reasoning, modeling and learning tasks (see Section 4.2.2).

FOL, also known as *predicate logic*, is a generalization of *propositional logic* (PL, see also the previous chapter). PL lacks the expressive power to capture a number of forms of reasoning. In particular, it cannot talk about *individuals* nor can it *quantify* over individuals, so as to say that *all* individuals have a certain property, or that *some* individual has. Where PL is capable of reasoning about *propositions*, FOL reasons over *objects*, *properties* of objects, and *relations* between objects.

Note that the formalization as given in this section is by no means the only possible one. Indeed, there are many notations and mathematical structures possible, as witnessed by the vast literature on logical methods in AI and computer science. Our treatment will describe the core components of FOL formalisms, though slightly biased towards the Horn and PROLOG setting described in Sections 4.2.2.1 and 4.2.2.2.

4.2.1.1 SYNTAX

The formal syntax of FOL is somewhat more complicated than that of PL. In analogue to the set of primitive propositions in PL, FOL's basis is formed by a (*first-order*) *vocabulary* Γ , which consists of *relation symbols* (or, *predicates*), *function symbols* (or, *functors*), and *constant symbols*. Relation and function symbols in Γ have some *arity*, which corresponds to the number of arguments they can take. A relation or function symbol m with arity α is denoted m/α . For example, the relation symbol `clear` takes one argument, and this is denoted `clear/1`. Relation or function symbols with arity 1, 2 and 3 are usually called *unary*, *binary* and *ternary*, respectively. Constants can be seen as *nullary* function symbols. Furthermore, there is an infinite supply of *variables*, which are usually denoted X , Y and Z . Variables and constants are both used to denote individuals. We will use the following notational convention in this book; relation and function symbols, as well as constants, start with a lowercase letter, for example `clear`, `onTop` and `floor`, whereas variables always start with an uppercase letter.

More complicated structures denoting individuals can be constructed from the function symbols. Formally, the set of *terms* is formed by starting from the constants and variables and closing off under function application:

DEFINITION 4.2.1 ► A **term** is defined inductively as: **i)** A constant is a term, **ii)** a variable is a term, **iii)** if f is an n -ary function symbol and t_1, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is a term. The terms t_1, \dots, t_n are the *arguments* of f , and **iv)** nothing else is a term.

Terms are syntactic structures that map eventually into the set of domain objects. Terms are used in formulas. The most basic unit is the *atomic formula* which is the smallest syntactic structure that can stand for a proposition:

DEFINITION 4.2.2 ► An **atom** is defined inductively as: **i)** If p is an n -ary predicate symbol and t_1, \dots, t_n are terms, then $p(t_1, \dots, t_n)$ is an atom and the t_1, \dots, t_n are the *arguments* of p , **ii)** if t_1 and t_2 are terms, then $t_1 = t_2$ is an atom, and **iii)** nothing else is an atom.

The set of variables occurring in an atom A is denoted $\text{var}(A)$. If A is an atom and $\text{var}(A) = \emptyset$ then A is called *ground*. For example, the atom `on(a, b)` is ground whereas `clear(X)` is not. As in PL, more complicated formulas can be formed by closing off under *connectives* such as negation, conjunction and disjunction, but FOL is closed off under another feature, that is *quantification*. The complete set of formulas is defined in the following definition:

DEFINITION 4.2.3 ► A well-formed **formula** is defined inductively as follows. **i)** An atom is a formula, **ii)** the special nullary predicates `true` and `false` are formulas, **iii)** if φ is a formula, then $\neg\varphi$ is a formula, called the negation of φ , **iv)** if φ and ψ are formulas, then $(\varphi \wedge \psi)$ (conjunction, or AND), $(\varphi \vee \psi)$ (disjunction, or OR) and $(\varphi \rightarrow \psi)$ (implication) are formulas, **v)** if φ and ψ are formulas, then $(\varphi \leftrightarrow \psi)$ is a formula (bi-implication, or equivalence), (We will often use an equivalent notation $(\varphi \equiv \psi)$), **vi)** if X is a variable and φ is a formula, then $(\exists X\varphi)$ is a formula, called an existential quantification, and $\exists X$ is called an existential quantifier, **vii)** if X is a variable and φ is a formula, then $(\forall X\varphi)$ is a formula, called an universal quantification, and $\forall X$ is called an universal quantifier, and **viii)** nothing else is a formula.

The part of a formula over which a quantifier exerts influence, is called the *scope* of a quantifier. The scope of $\exists X$ in $(\exists X\varphi)$ and $\forall X$ in $(\forall X\varphi)$ is both φ . A variable X in a formula is *bound* iff it is in the scope of a quantifier or it occurs in $\exists X$ or $\forall X$. The occurrence of a variable is *free* in a formula iff it is not bound in the formula. A *sentence* (or, closed formula) is a formula containing no occurrences of variables that are free in the formula. *Nesting* of quantifiers is allowed under the common assumption that variables that are bound by one quantifier are not reused for another quantification in that same scope. The *order* of nested quantifiers is very important. The formula $\forall X\exists Y \text{ larger}(X, Y)$ means that for every block one can find another that is larger, whereas $\exists Y\forall X \text{ larger}(X, Y)$ means that there is a block that is larger than all other ones.

In order to simplify notation, we will omit parentheses when possible. Furthermore, we will use $t_1 \neq t_2$ as an abbreviation of $\neg(t_1 = t_2)$ and we will use $\exists X_1, \dots, X_n\varphi$ and $\forall X_1, \dots, X_n\varphi$ as abbreviations of $\exists X_1, \dots, \exists X_n\varphi$ and $\forall X_1, \dots, \forall X_n\varphi$. Yet another notation we often use for *some* vector of variables is $\forall \vec{X}\varphi(\vec{X})$.

4.2.1.2 SEMANTICS

There are many forms of semantics. Two important ones are *Tarskian* and *Herbrand* semantics. In this section we will deal with the classical Tarskian version, which interprets formulas in terms of correspondence with objects in a structure. In Section 4.2.2.1 we will describe the more restricted Herbrand semantics which lets formulas of the language stand for themselves. A third form of semantics is *Kripke* semantics, or *possible worlds* semantics, which is used for *modal* logics and we will only briefly describe its main characteristics in Section 4.2.2.3 for its intimate relationship with POMDPs¹².

For philosophers semantics is about the relationship between models and the real world. For logicians and in this book, semantics is about the relationship between *statements* and *models*. Semantics determines the *truth* of formulas with respect to a *structure*. The definition of a structure (sometimes called an *interpretation*) will tell us **i)** what collection of things the universal quantifier (\forall) ranges over (the *domain* or *universe of quantification*) and **ii)** what the other parts – the relation and function symbols – denote with respect to the domain of the structure. In the following we will provide a formal connection between the language (i.e. the syntax) and structures (i.e. the semantics). Let Γ be a logical vocabulary as defined in the previous section.

DEFINITION 4.2.4 ► A Γ -**structure** \mathcal{A} consists of a non-empty domain $\mathcal{D} = \text{dom}(\mathcal{A})$, an assignment of a k -ary relation $p^{\mathcal{A}} \subseteq \mathcal{D}^k$ to each k -ary relation symbol p of Γ , an assignment of a k -ary function $f^{\mathcal{A}} : \mathcal{D}^k \rightarrow \mathcal{D}$ to each k -ary function symbol f of Γ , and an assignment of a member $c^{\mathcal{A}}$ of the domain to each constant symbol c . $p^{\mathcal{A}}$, $f^{\mathcal{A}}$, and $c^{\mathcal{A}}$ are called the *denotations* of P , f , and c , respectively, in \mathcal{A} .

Suppose that Γ consists of one binary relation symbol r . In this case, the Γ -structure \mathcal{A} is essentially a directed graph. The domain of the structure is formed by the nodes in the graph and each connection between nodes corresponds to the relation r between nodes. Let n_1 and n_2 be two nodes of the graph. There is an edge between nodes n_1 and n_2 iff $(n_1, n_2) \in r^{\mathcal{A}}$. A relational structure does not provide all means to give semantics

¹²Possible worlds semantics is also of interest when talking about denotations and when representing *partial states* or *belief states*.

to formulas. For example, a formula $\exists X \text{clear}(X)$, saying that there is *some* clear block, should be true in all structures in which there is some individual in $\text{dom}(\mathcal{A})$ that is also in $\text{clear}^{\mathcal{A}}$. A *valuation function* provides an interpretation of the variables occurring in logical formulas:

DEFINITION 4.2.5 ▶ A **valuation function** V on structure \mathcal{A} is a function from variables to \mathcal{D} . $V^{\mathcal{A}}(c) = c^{\mathcal{A}}$ for constant symbol c and $V^{\mathcal{A}}(f(t_1, \dots, t_k)) = f^{\mathcal{A}}(V^{\mathcal{A}}(t_1), \dots, V^{\mathcal{A}}(t_k))$ for function symbol f .

Additionally, it will prove useful to define the *denotation* of terms, which corresponds roughly to the inverse of the mapping from syntactic elements to semantic structures.

DEFINITION 4.2.6 ▶ Let α be a structure and \mathcal{V} a valuation function. The **denotation** of term t , denoted $\llbracket t \rrbracket_{\mathcal{A}, \mathcal{V}}$ is defined by: **i)** if X is a variable, then $\llbracket X \rrbracket_{\mathcal{A}, \mathcal{V}} = \mathcal{V}(X)$, and **ii)** if t_1, \dots, t_n are terms, and f is an n -ary function symbol then $\llbracket f(t_1, \dots, t_n) \rrbracket_{\mathcal{A}, \mathcal{V}} = f^{\mathcal{A}}(d_1, \dots, d_n)$ where $d_i = \llbracket t_i \rrbracket_{\mathcal{A}, \mathcal{V}}$.

Now everything is in place to relate the syntactic elements (formulas) to semantic structures (interpretations). The following definition formalizes this connection in a *compositional* way, by making use of the structure of formulas.

DEFINITION 4.2.7 ▶ **Truth in a structure \mathcal{A} under valuation \mathcal{V}** proceeds by induction on formula structure. If \mathcal{V} is a valuation function, X is a variable, and $d \in \text{dom}(\mathcal{A})$, let $V[X/d]$ be the valuation \mathcal{V}' such that $\mathcal{V}'(Y) = \mathcal{V}(Y)$ for every variable Y except X , and $\mathcal{V}'(X) = d$.

$$\begin{aligned}
 (\mathcal{A}, \mathcal{V}) \models p(t_1, \dots, t_k) & \quad \text{where } p \text{ is a } k\text{-ary relation and } t_1, \dots, t_k \text{ are terms,} \\
 & \quad \text{iff } (\mathcal{V}(t_1), \dots, \mathcal{V}(t_k)) \in p^{\mathcal{A}}; \\
 (\mathcal{A}, \mathcal{V}) \models (t_1 = t_2) & \quad \text{where } t_1 \text{ and } t_2 \text{ are terms,} \\
 & \quad \text{iff } \mathcal{V}(t_1) = \mathcal{V}(t_2) \\
 (\mathcal{A}, \mathcal{V}) \models \neg \varphi & \quad \text{iff } (\mathcal{A}, \mathcal{V}) \not\models \varphi \\
 (\mathcal{A}, \mathcal{V}) \models \varphi_1 \wedge \varphi_2 & \quad \text{iff } (\mathcal{A}, \mathcal{V}) \models \varphi_1 \text{ and } (\mathcal{A}, \mathcal{V}) \models \varphi_2 \\
 (\mathcal{A}, \mathcal{V}) \models \varphi_1 \vee \varphi_2 & \quad \text{iff } (\mathcal{A}, \mathcal{V}) \models \varphi_1 \text{ or } (\mathcal{A}, \mathcal{V}) \models \varphi_2 \\
 (\mathcal{A}, \mathcal{V}) \models \varphi_1 \rightarrow \varphi_2 & \quad \text{iff } (\mathcal{A}, \mathcal{V}) \models \varphi_2 \text{ or } (\mathcal{A}, \mathcal{V}) \models \neg \varphi_1 \\
 (\mathcal{A}, \mathcal{V}) \models \exists X \varphi & \quad \text{iff } (\mathcal{A}, \mathcal{V}[X/d]) \models \varphi \text{ for some } d \in \text{dom}(\mathcal{A}) \\
 (\mathcal{A}, \mathcal{V}) \models \forall X \varphi & \quad \text{iff } (\mathcal{A}, \mathcal{V}[X/d]) \models \varphi \text{ for all } d \in \text{dom}(\mathcal{A})
 \end{aligned}$$

Note that $\forall X \varphi$ is an abbreviation of $\neg \exists X \neg \varphi$. Furthermore, a valuation does not affect the truth of a sentence. We will usually omit the valuation function when possible.

A formula φ is *valid in a structure* (or, is *true* in a structure) \mathcal{A} , denoted $\mathcal{A} \models \varphi$, if $(\mathcal{A}, \mathcal{V}) \models \varphi$ for all valuations \mathcal{V} . In this case, we say that \mathcal{A} is a *model* of φ and that \mathcal{A} *satisfies* φ . It is *satisfiable* if $(\mathcal{A}, \mathcal{V}) \models \varphi$ for *some* structure \mathcal{A} and *some* valuation \mathcal{V} . It is *valid* if $\mathcal{A} \models \varphi$ for *all* structures \mathcal{A} , denoted $\models \varphi$. A valid formula is called a *tautology*. A formula φ *entails* a formula ψ (or, ψ is a *logical consequence* of φ), denoted $\varphi \models \psi$, iff for every structure \mathcal{A} such that $\mathcal{A} \models \varphi$, we have $\mathcal{A} \models \psi$. In other words, φ entails ψ if every model of φ is a model of ψ . A structure \mathcal{A} is a model for a set of formulas Φ if \mathcal{A} is a model of every formula in Φ .

Any satisfiable set of formulas has (infinitely) many models. This means that the formulas which properly describe some particular, 'intended' world of interest at the same time describe many other worlds. The set of models of a formula (or a set of formulas) is

called the *denotation of a formula*.

DEFINITION 4.2.8 ▶ Let Γ be a logical vocabulary and $\mathcal{A}(\Gamma)$ be the set of all Γ -structures. Let φ be a formula of the language, and let \mathcal{V} be a valuation function. The **denotation of the formula** φ , denoted $\llbracket \varphi \rrbracket_{\mathcal{A}, \mathcal{V}}$, is the set $\mathcal{A}' \subseteq \mathcal{A}(\Gamma)$ such that for all $\mathcal{A} \in \mathcal{A}'$, $(\mathcal{A}, \mathcal{V}) \models \varphi$. In other words, \mathcal{A}' consists of all models of φ .

Denotations are useful for reasoning about the sets of models of a formula or set of formulas. If it is clear from the context, we will omit the subscripted Γ -structure \mathcal{A} and valuation function. Definition 4.2.7 can now be cast in set-theoretic operations. For example, let φ and ψ be formulas, then $\llbracket \varphi \wedge \psi \rrbracket = \llbracket \varphi \rrbracket \cap \llbracket \psi \rrbracket$. The set-theoretic interpretation will prove useful when reasoning about the sets of structures (e.g. sets of states of an RMDP) the formulas generalize over.

4.2.1.3 ENTAILMENT, SOUNDNESS AND COMPLETENESS

The syntax of FOL provides us with a formal system to specify and represent relational worlds. The semantics formalizes mathematical structures that give *meaning* to the symbols of the language. Two main strands in the application of logic are *model theory* and *proof theory*. Model theory is concerned with attributing meaning (i.e. truth value) to sentences (i.e. well-formed formulas) in a first-order language. Proof theory is concerned with (*deductive*) reasoning with formulas in a language, i.e. how to do common-sense (logical) reasoning within a logical system.

FOL domains can be *axiomatized* using a set of sentences called *axioms*, together with *rules* that enable the derivation of new formulas from old. This permits the construction of *proofs* of sentences from sets of sentences called *premises*. An *axiom system* AX consists of a set of *logical axioms* and a set of *inference rules*. The logical axioms are formulas, and inference rules specify how a formula may be inferred from other formulas. A standard inference rule is *modus ponens*, which states that, from φ and $\varphi \rightarrow \psi$ we may infer ψ .

A *proof* of a formula ψ given a formula φ is a finite sequence of formulas such that the last formula is ψ and each formula is either a logical axiom, φ itself, or a formula inferred from previous formulas using one of the inference rules. A formula ψ is *provable* from a formula φ using axiom system AX , denoted $\varphi \vdash_{AX} \psi$ iff there is a proof of ψ given φ . We usually omit AX from \vdash_{AX} when it is clear from the context which axiom system is used. A formula φ is *consistent* iff there is no formula ψ such that both $\varphi \vdash \psi$ and $\varphi \vdash \neg\psi$.

A knowledge base KB is a set of sentences where the *explicit beliefs* consists of the KB itself, and the *implicit beliefs* are all logical consequences¹³ of KB. That is, the explicit beliefs are those sentences that are immediately accessible, whereas the implicit beliefs require a proof, or *deduction* steps, to become available. There are several ways to formulate the deduction task itself. Let KB be the finite set of sentences $\{\varphi_1, \dots, \varphi_n\}$. Then one can formulate it as **i)** $KB \models \psi$, **ii)** $\models (\varphi_1 \wedge \dots \wedge \varphi_n) \rightarrow \psi$, **iii)** $KB \cup \{\neg\psi\}$ is not satisfiable, and **iv)** $KB \cup \{\neg\psi\} \models \text{false}$. Regardless of the definition of the deduction task, a general proof mechanism takes a set of sentences KB, an axiom system AX and a desired conclusion ψ . The first step is to take a formula φ_0 from KB, apply some axiom AX_i from AX , and

¹³Interestingly, this is a phenomenon frequently occurring in logics and agents. The fact that an agent knows all logical consequences of its knowledge base is called *logical omniscience*. This is often not realistic in the practical sense. For example, even though I know some mathematics, I would not claim to *know* the validity of Fermat's last theorem, the millionth digit of π , let alone whether $P = NP$ is true.

derive a new formula φ_1 . This process continues as

$$\varphi_0 \xrightarrow{AX^0} \varphi_1 \xrightarrow{AX^1} \dots \xrightarrow{AX^n} \psi \quad (4.3)$$

where AX^k is the axiom used in the k th deduction step in the proof. Many distinct axiom systems and proof procedures exist, differing mainly in **i)** the specific subset of FOL that is used, **ii)** the sentences in the axiom system, and **iii)** the details of the proof procedure (e.g. how and when specific axioms and premises are chosen and used). There are many proof systems, such as based on *natural deduction*, *sequent calculus* and *semantic tableaux*. In Section 4.2.2.1 we will describe one particular implementation in some more detail, one that is based on **i)** the restricted language of *Horn clauses*, **ii)** *resolution*, and **iii)** a proof procedure based on a linear ordering of axioms in KB and *depth-first proof trees*.

An intimate relationship exists between proof and model theory. An axiom system AX is *sound* iff for all formulas φ and ψ , it is the case that $\varphi \vdash \psi$ implies $\varphi \models \psi$ (i.e. whenever it produces something, it is entailed) and it is *complete* iff $\varphi \models \psi$ implies $\varphi \vdash \psi$ (i.e. whenever it is entailed, it will be produced). Many sound and complete axiom systems exist for FOL. The *computational complexity* of logical systems usually resides in the reasoning process, which – in part – depends on which types of expressions are allowed. Entailment for *full* FOL is *not* decidable; the non-valid sentences are not recursively enumerable, although the valid sentences are. Checking whether a particular interpretation is a model of a theory (i.e. *model checking*) however, is decidable. Poole *et al.* (1998, pages 175–177) provide a summary of complexity results, showing that deciding entailment for FOL, *clausal logic*, *Horn clauses* and *definite clauses* is undecidable, whereas restricted logics such as functor-free FOL, *Datalog* and propositional clauses are decidable, but NP-hard. A rule of thumb is that more restrictions on the *form* of the formulas (e.g. disallowing certain connectives or quantifier nesting) enable more efficient computational methods for theorem proving in the average case, or even make the problem decidable. Restricted formalisms are the topic of the following sections.

4.2.2 Fragments and Extensions of FOL

The tradeoff between *expressiveness* and *tractability* is one of the biggest challenges when using logical knowledge representation formalisms (Brachman and Levesque, 2004, Chap. 16). Indeed, from a *representational* point of view, it is highly desirable to use an expressive formalism such as full FOL (or even its extensions). However, from a *reasoning* (or computational) point of view, this might lead to serious problems. There are many reasoning tasks that can easily be formulated in terms of FOL entailment but they can often be solved more efficiently by special-purpose methods because of (syntactic) restrictions on the KB or on φ . In a computational context such as relational RL, important features of logical formalisms are the following:

1. **Expressivity and Complexity of Structures.** Allowing all connectives and quantifiers, possibly augmented with various *modal operators* (see Section 4.2.2.3) allows for very powerful and expressive syntax. *Normal forms* (e.g. DNF, CNF) often exist, though they blow up the syntactic representation. Semantic structures may also differ in complexity. Rich semantics involving for example *functions* and *temporal structures* (e.g. *time*), thereby extending the standard relational structures, enable rich models, though they make it necessary to extend the syntax to represent them.

2. **Reasoning Complexity.** The complexity of deciding entailment is at the heart of logical formalisms. This complexity varies between formalisms (Poole *et al.*, 1998, pages 175–177), and a key point is to provide means to *restrict* and *bias* the language and the reasoning process to make it tractable. This includes a move from purely *declarative* to *procedural* knowledge. *“Rather than expecting a theorem prover to do all the work, all (significant) inferences should be made before run time and stored as compiled knowledge”* (Omar, 1994, p. 166). Most systems restrict the syntax and provide specialized procedures (axiom systems plus control over theorem proving) to make reasoning more efficient. We will see various examples throughout this book.
3. **Comprehensibility and Transfer.** Comprehensibility is often not considered explicitly when logical methods are employed (but see Džeroski and Lavrac, 2001b, Sec. 15.6). However, it is of great importance for *ease of modeling*, *readability* of formalizations and *interpretation* of computed results. *Bioinformatics* (Orengo *et al.*, 2003) is an area where comprehensibility is very important. *Graphical* representations are often preferred over logical formulas, which explains the popularity of (logical) Bayesian networks. *Declarative* knowledge is often more comprehensible than procedural. Many systems add *syntactic sugar* to the language to increase comprehensibility, without changing the formal expressiveness of the language.

A related aspect is *transfer*, i.e. the *reuse* of formalizations or computed results in other domains, or related problems. Formalisms that are more declarative in nature are often more amenable to transfer. Declarative knowledge supports better the *locality* of knowledge, and allows for easier transfer of only parts of the system and for easier incorporation of prior knowledge.

4. **Efficient Data Structures and Algorithms.** Logical reasoning is computationally demanding and various aspects can be made more efficient. This does not change worst-case behavior of reasoning (i.e. the complexity class to which it belongs), though the average behavior will. First of all, *storage* of formulas, axioms and examples can greatly benefit from compact representations such as (first-order) *trees*, *ADDs* and *BDDs*, in order to avoid redundancies (e.g. see Section 4.2.3). Second, *manipulating* syntactic expressions can be translated to efficient *operations* on these compact representations. Last, efficiency in reasoning can be increased by making use of smart algorithms such as *caching* and *query packs* (Blockeel *et al.*, 2000b) and *tabling* (see Chapter 6). Furthermore, *language* and *search biases* can be employed to make (inductive and deductive) reasoning more efficient (see Section 4.3.2). These biases can also be employed to restrict the space of models considered (see Section 4.3.3).
5. **Approximations.** For many complex tasks, logical representation and reasoning can be too complex to perform completely. In addition, one might be satisfied by a reasonable, instead of optimal, performance. Approximations can be used in the problem representation, in the reasoning process and in the sampling of problem data. For example, instead of generating all possible structures, one can sample tests (Srinivasan, 1999), and instead of a complete set of sentences one can use linear combinations of so-called basis functions (Sanner and Boutilier, 2006). Especially

in relational RL, approximations can aid in coping with uncertainty, dynamic worlds and keeping solution structures compact.

These five issues have many interdependencies. For example, the trade-off between (1) and (2) is well-known. Furthermore, more expressive formalisms (1) are often (but not always) more comprehensible (3), because they are usually more close to the structure of natural language expressions. Efficient data structures (4) are often very useful *internally* in storage and reasoning mechanisms (2), though their compactness may require a translation step to present the computed output in comprehensible form (3). Furthermore, efficient data structures (4) make implementation more complex, though they speed up reasoning on the logical level (2). Approximations (5) play a role in all first four and make things easier in computational terms, at the expense of the quality or completeness of computed solutions.

Overall, in many computational domains where logical methods are applied, an important guideline is that *simplicity is key*. In logical ML, relational RL, planning and other tasks where the burden lies upon computational aspects, keeping the representation and reasoning mechanisms as simple as possible, is *vital* for actually computing anything. In the following we provide one such simpler logical method in a little more detail – *clausal logic* and *logic programming* – and additionally discuss briefly some other formalisms. Reasons for doing this are twofold. First, it shows some types of variations of FOL and how they relate to each other and FOL. Second, these are the logical systems typically employed for logical ML and action languages and, in particular, for all the relational RL systems we describe in Chapters 5 to 7.

4.2.2.1 CLAUSAL LOGIC

One special form of (computational) logic we will be concerned with is *clausal logic*, and the intimately related programming language PROLOG. Both systems form the basis for many first-order approaches to ML (see Section 4.3) and action theories (see Section 4.4), as well as the large majority of work in relational RL, including the material in Chapters 5 to 7. Horn logic is a useful (syntactic) restriction of FOL that allows for efficient reasoning and ML, due to the fact that *resolution* becomes manageable as a first reasoning principle. We define clausal logic and resolution in this section and postpone a redefinition of the underlying semantics to the next section.

A *clause* is a disjunction of (positive and negative) *literals*, which are atoms defined in the previous sections, preceded by a prefix of universal quantifiers ranging over the variables occurring in the literals. Formally, a clause has the following *disjunctive normal form* (DNF) form, $L_1 \vee \dots \vee L_n$, where L_1, \dots, L_n are literals. Clauses are (implicitly) universally quantified, such that this clause should be read as $\forall X_1, \dots, X_k (L_1 \vee \dots \vee L_n)$. The most general form of a clause is $\forall X_1, \dots, X_n (\neg B_1 \vee \dots \vee \neg B_i \vee A_1 \vee \dots \vee A_j)$ which is equivalent to $\forall X_1, \dots, X_n ((B_1 \wedge \dots \wedge B_i) \rightarrow (A_1 \vee \dots \vee A_j))$. The commonly used notation for a clause is: $A_1, \dots, A_i \leftarrow B_1, \dots, B_j$. A *Horn clause* is a clause with at most one positive literal. In the equivalent notation we obtain

$$A \leftarrow B_1, \dots, B_j$$

where A is called the *head* of the clause and B_1, \dots, B_j the *body*. This type of notation is sometimes called *program notation* and it provides a syntactic link to the PROLOG language

(see Section 4.2.2.2). An equivalent *set notation* represents this clause as $\{\neg A \vee B_1 \vee \dots \vee B_j\}$. A *definite clause* is a Horn clause with exactly one positive literal. A set of clauses is called a *clausal theory* and it represents the conjunction of its clauses. A clause or clausal theory is called *function-free* if it contains only variables as terms (this also means no constants). A *Datalog clause* is a definite clause that contains no function symbols of non-zero arity. A *goal clause*, also called a *query*, is a Horn clause containing no positive literals. The *empty clause* is denoted \square .

DEFINITION 4.2.9 ► A **knowledge base in clausal form** is a set $KB = \{\varphi_i, \dots, \varphi_n\}$ of definite clauses, where each $\varphi_i, i = 1 \dots n$ can take either the form of a **fact** or a **rule**. A fact is a ground atom, stating a basic truth in the domain. A rule is a definite clause, extending the vocabulary, expressing new relations in terms of basic facts or other relations.

Horn clauses are useful for many knowledge representation tasks such as *relational databases*. For example, consider a database about family relations. Examples of *atomic facts* are `parentOf(greet, martijn)`, `sonOf(martijn, wim)` and `sisterOf(jose, martijn)`. A rule about family relations would be $\forall X \forall Y ((\text{parentOf}(X, Y) \wedge \text{woman}(X)) \rightarrow \text{motherOf}(X, Y))$ which, in program notation is equivalent to `motherOf(X, Y) ← parentOf(X, Y), woman(X)`. In fact, there are many similarities between the terminologies and techniques of deductive databases and logic programming (see e.g. Džeroski and Lavrac, 2001a; Wrobel, 2001).

In order to apply resolution for reasoning with definite clauses, we need some additional machinery. Because of quantification in clauses, they can be instantiated for specific (ground) instances. This is done by *substitution* and *unification*.

DEFINITION 4.2.10 ► A **substitution** θ is a finite set $\{X_1/t_1, \dots, X_n/t_n\}$ where each pair $X_i/t_i, (i = 1 \dots n)$ consists of a variable X_i and a term t_i such that $X_i \neq t_i$ and $X_i \neq X_j, i \neq j$. The empty substitution is denoted ϵ . A substitution $\theta = \{X_1/t_1, \dots, X_n/t_n\}$ is **applied** to an atom A , denoted $A\theta$, by replacing all occurrences X_i in A by t_i . A **grounding substitution** θ for an atom A is such that $A\theta$ is ground.

A **unifier** for literals A and B is a substitution θ such that – assuming $\text{var}(A) \cap \text{var}(B) = \emptyset$ which can be guaranteed by variable renaming – $A\theta \equiv B\theta$. The **most general unifier** (mgu), for literals A and B is denoted $\text{mgu}(A, B)$. If θ is a most general unifier, then every other unifier θ' can be written as $\theta' \equiv \theta\theta''$ where θ'' is a unifier.

Unifiers and mgu's can be extended to more complex formulas (e.g. clauses, or conjunctions of atoms) in a straightforward way. Note the similarities with valuation functions.

As mentioned, the main reasoning principle that is useful for Horn theories, is *resolution*. Standard (propositional) resolution operates on disjunctions and derives $A \vee C$ from $B \vee A$ and $\neg B \vee C$. For a first-order theory in Horn form, the proof method follows the general structure depicted in Equation 4.3, where the axiom system AX consists only of the following rule:

DEFINITION 4.2.11 ► Let KB be a theory in clausal form. **Resolution** comprises a single inference rule for clausal logic, and is formally defined as:

$$\frac{\leftarrow A_1, \dots, A_{i-1}, A_i, A_{i+1}, \dots, A_m \quad B_0 \leftarrow B_1, \dots, B_n}{\leftarrow (A_1, \dots, A_{i-1}, B_1, \dots, B_n, A_{i+1}, \dots, A_m)\theta} \quad (4.4)$$

where **i)** A_1, \dots, A_m are literals, **ii)** $B_0 \leftarrow B_1, \dots, B_n$ is a (renamed) definite clause in KB

($n \geq 0$), and **iii**) $\text{mgu}(A_i, B_0) = \theta$.

The *premises* of the rule are a goal clause and a definite clause. Both are separately universally quantified, such that their scopes are disjoint. The *conclusion* contains only one quantification, making it necessary that the sets of variables of the premises are disjoint, which can always be satisfied by renaming (because they are all bound).

The two basic types of reasoning, or, *questions* one would like to ask are YES/NO-questions and WHICH-questions. The common form of questions is that of a headless clause $\leftarrow B_1, \dots, B_n$, and is called a *query*. An example YES/NO-question is `sonOf(martijn, greet)?` which asks whether this relation is entailed by KB. More interesting is a WHICH-question such as $\exists X \text{parentOf}(X, \text{martijn})?$ which asks for all persons that are the parent of `martijn`, i.e. a grounding substitution θ such that $\text{parentOf}(X, \text{martijn})\theta$ is entailed by KB.

Formally, Let KB be a knowledge base in clausal form, let AX consist of the resolution rule, and let $\leftarrow B_1, \dots, B_n$ be a query. The reasoning task is now to decide whether $\text{KB} \vdash_{AX} \exists X_1 \dots X_k (B_1, \wedge \dots \wedge B_n)$. Rewriting this results in $\text{KB}, \neg \exists X_1 \dots X_k (B_1, \wedge \dots \wedge B_n) \vdash_{AX} \perp$, and then $\text{KB}, \forall X_1 \dots X_k (\neg B_1 \vee \dots \vee \neg B_n) \vdash_{AX} \perp$, where \perp stands for false, which in this case coincides with \square . Finally, this is equivalent to adding the query to the KB and deriving \square , i.e. $\text{KB} \cup \{\leftarrow B_1, \dots, B_n\}$ leads to \square using resolution. Summarizing, resolution proves that $\text{KB} \models B\theta$, by proving $\text{KB} \wedge \neg B\theta$ is unsatisfiable, i.e. by deriving the empty clause \square . We will make this more explicit in the next section.

Resolution is *refutation-complete* which means that if a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction. Resolution cannot be used to generate all logical consequences of a set of sentences, but it can be used to establish that a given sentence is entailed by a set of sentences. In the next section we will see that there is a little more to say about *negation* and resolution.

4.2.2.2 LOGIC PROGRAMMING AND PROLOG

Logic programming (LP) (Lloyd, 1991; Sterling and Shapiro, 1994; Bratko, 2001) as a technique comes very close to the idea of declarative knowledge engineering and reasoning. The ideal case is to express knowledge declaratively in a formal language (such as FOL) and run an inference process on that knowledge to derive desirable or requested statements. Although logic has been used as a fundamental tool in AI and KR since almost the beginning, the use of logic directly as a programming language, is more recent. The central idea in LP is that a *declarative* statement of the form $A \text{ if } B \text{ and } C \text{ and } D$ can be read *procedurally* as *to solve A, solve B, C and D*. In other words, proof mechanisms can also be used to *compute* something. The LP language PROLOG is a practical realization of this idea. In its basic form, it provides a *declarative programming* language, though most implementations provide a procedural interpretation. Furthermore, it provides an *interactive* environment in which a user can interact with the program by querying it.

A *program clause* is a clause of the form $A \leftarrow L_1, \dots, L_n$ where A is an atom and each of L_1, \dots, L_n is a positive or a negative literal. Negative literals are written in the form `not L` to avoid confusion with logical negation (\neg). A *logic program* consists of a set of program clauses. A set of program clauses with the same head (predicate symbol and arity) form together a *predicate definition*. A set of definite clauses is called a *definite logic program*.

A notational convention in PROLOG is that facts are represented as `r(c1, ..., cn)`.

clear(a).	above(X, Y) :-	height(X, H) :-
clear(b).	on(X, Y).	onTop(Y, X),
	above(X, Y) :-	height2(Y, H).
	on(X, Z),	
on(a, e).	above(Z, Y).	height2(X, 0) :-
on(b, d).		X == floor.
on(d, c).	inTower1(X) :-	height2(X, H) :-
on(e, floor).	not on(X, floor).	on(X, Y),
on(c, floor).		height2(Y, H2),
	inTower2(X) :-	H is H2 + 1.
block(a).	on(X, Y),	tower([floor]).
block(b).	block(Y).	tower([X Xs]) :-
block(c).		block(X),
block(d).	onTop(X, Y) :-	tower(Xs).
block(e).	clear(X),	
	above(X, Y).	

Figure 4.6: A sample PROLOG program, with on the left the extensional clauses (facts) and on the right the intensional clauses (predicate definitions).

where r is an n -ary relation symbol. Clauses are represented as usual, but with $:-$ replacing the arrow notation and a dot following each clause. Furthermore, PROLOG includes some syntactic sugar for arithmetic and lists. For example, the expression X is $Y + 1$ denotes that X is assigned the value of Y increased by 1, and the expression $[a, b, c]$ denotes an ordered list containing three elements. Figure 4.6 shows some examples. In the following we first restrict our exposition to definite programs. The set of logical consequences of a program is infinite. Therefore the user is expected to *query* the program selectively for various aspects of the intended model. This is in analogy with databases.

Let us turn to semantics now. Because of the fact that PROLOG is a computational method for reasoning with clauses and clausal logic is a proper subset of FOL, the FOL semantics as defined in Section 4.2.1.2 can be used to give semantics to PROLOG programs too. Nevertheless, logic programming formalisms are usually given semantics in a more restricted fashion than the full FOL setting. The Tarskian semantics defined there allowed for infinitely many models (interpretations, structures) for any given formula, which was induced by separate symbols in the language and in the structures. The simpler *Herbrand* setting used here restricts the possible interpretations to include only elements of the language itself, i.e. the predicate and function symbols and the constants in the language. Interpretations are constructed solely from ground atoms that can be built in the language.

DEFINITION 4.2.12 ▶ Let Γ be a logical vocabulary and let P be a logic program in Γ . The **Herbrand universe** of P , denoted $HU(P)$, is the set of all ground terms that can be constructed from the constants and function symbols appearing in P . The **Herbrand base**, denoted $HB(P)$, is the set of all ground goals that can be constructed from the predicates in P and the terms in $HU(P)$. An **interpretation** i of a logic program is a subset of the Herbrand base, i.e. $i \subseteq HB(P)$.

Note that due to the application of function symbols, the Herbrand universe can be infinite.

The Herbrand base is infinite if the Herbrand universe is. Notice the differences with the semantics as defined in Section 4.2.1.2. There is no mapping from syntactic elements to semantic elements; each interpretation is represented using the elements in the syntax.

EXAMPLE 4.2.1 ► An interpretation corresponds to a possible state of the world. It consists of all ground facts that hold in that state, and is therefore represented as a set. For example, $\{\text{on}(\text{a}, \text{b}), \text{clear}(\text{a}), \text{on}(\text{b}, \text{floor}), \text{on}(\text{c}, \text{floor}), \text{clear}(\text{c}), \text{clear}(\text{floor})\}$ is a BLOCKS WORLD state. Notice that the state space definition of RMDPs (see Definition 4.1.1) is given by all Herbrand interpretations.

Slightly more complicated are functors and recursive terms. For example, consider the set of natural numbers with base case $\text{nat}(0)$ and inductive definition $\text{nat}(\text{s}(X)) \leftarrow \text{nat}(X)$. The Herbrand universe includes the infinite set of terms $\text{nat}(\text{nat}(\text{nat}(\dots(\text{nat}(0))\dots))\dots)$. This also renders the Herbrand base and the set of Herbrand interpretations infinite. RMDPs defined in a language with this $\text{nat}/1$ definition can have infinite state spaces.

Analogous to the FOL definition of satisfiability given a set of formulas, one can define when an interpretation is a model of a program:

DEFINITION 4.2.13 ► An interpretation i is a **model** of a definite logic program P if for each ground instance of a clause $A \leftarrow B_1, \dots, B_n$ in P , A is in i if B_1, \dots, B_n are in i .

Herbrand's theorem states that a set of clauses is satisfiable if and only if its Herbrand base is. Thus, a Herbrand interpretation I is a model for a clause $C \equiv H \leftarrow B$ iff for all substitutions θ such that $C\theta$ is ground, $B\theta \subset I$ implies that $H\theta \cap I \neq \emptyset$. An interpretation is a model of a clausal theory T (e.g. a program P) iff it is a model of all clauses in T (e.g. P). When writing a logic program, one has a particular *meaning* (i.e. the interpretations it describes) in mind. This is captured by the *declarative meaning* of the program:

DEFINITION 4.2.14 ► The model obtained as the intersection of all models is known as the **minimal model** and denoted $M(P)$. The minimal model is the **declarative meaning** of a definite logic program.

Note that the Herbrand base of a definite program is always a Herbrand model of that program. However, the *intended model* – also called the *least Herbrand model* – is the model the programmer has in mind when writing – declaratively – a program. It consists of all ground atomic logical consequences of the program P , that is $\{A \in \text{HB}^P \mid P \models A\}$. If the intended model is not within the set of models of the program, then the program is said to be *incorrect*.

The construction of the minimal model of a program P is performed using a fixed point of a *forward chaining* procedure on interpretations of P :

DEFINITION 4.2.15 ► Let P be a definite program. Let $g(P)$ be the set of all ground instances of clauses in P . T^P is a function on Herbrand interpretations of P defined as

$$T^P(I) = \{A_0 \mid A_0 \leftarrow A_1, \dots, A_n \in g(P) \wedge \{A_1, \dots, A_n\} \subseteq I\}$$

$M(P)$ is the *least fixed point* of T^P , i.e. the least Herbrand interpretation such that $T^P(M(P)) = M(P)$. In other words, $M(P) = T^P \uparrow v$, where **i)** $T^P \uparrow 0 = \emptyset$, **ii)** $T^P \uparrow (i + 1) = T^P(T^P \uparrow i)$, and **iii)** $T^P \uparrow v = \bigcup_{i=1}^{\infty} T^P \uparrow i$.

For additional details and proofs, see for example (Nilsson and Maluszinski, 1995). Unless stated otherwise, we will reserve the term *interpretation* for Herbrand interpretations of a given clausal KB, or equivalently, a PROLOG program.

EXAMPLE 4.2.2 ▶ Let program P consist of $r(X) : - p(X)$ and $p(X) : - q(X)$ and let $i = \{q(a), q(b)\}$. Then $T^P(i) = \{q(a), q(b), p(a), p(b)\}$, $T^P(T^P(i)) = (T^P)^2(i) = \{q(a), q(b), p(a), p(b), r(a), r(b)\}$ and finally $(T^P)^n(i) = (T^P)^2(i) = M(P)$, the minimal Herbrand model of P . If one adds $t(X, Y) : - r(X), q(Y)$, the minimal model $M(P)$ is *completed* by adding $\{t(a, b), t(b, a), t(b, b), t(a, a)\}$.

As we have seen, the syntax and semantics of PROLOG programs can be framed into a restricted FOL setting. Syntactic restrictions cast the language into clausal formulas, whereas the accompanying Herbrand semantics can be seen as a restricted form of the standard (Tarskian) semantics in which basic (ground) language expressions stand for themselves. We will now turn to the *reasoning* part of the PROLOG language, which can be seen as a practical implementation of the resolution procedure outlined in the previous section, augmented with several alterations for negation and control of reasoning.

A PROLOG interpreter does theorem proving – or, answering queries – using resolution. In fact, the method we present here, is an extension of the resolution rule that can deal with negative literals as well. But let us first consider definite programs. In Equation 4.4, the goal clause in the premise may contain several atomic formulas that unify with the head of a program clause. Therefore some *selection function* is needed to (deterministically) select which goal atom is selected first. One can use various selection rules using various kinds of information (i.e. contextual information not present in the goal clause) but usually a simple *left-to-right* selection of goals is chosen. The inference rule in Equation 4.4 is a restricted form of the *resolution principle* for clauses. Here it is formalized for definite clauses. It is known under the name of *SLD-resolution* (which stands for *Linear resolution for Definite clauses with Selection function*, sometimes denoted \vdash_{SLD}). Let a definite goal clause G_0 be of the form $\leftarrow A_1, \dots, A_m$. From this goal clause, a subgoal A_i is selected (if possible) using the selection rule. A new goal clause is constructed (if possible) from a (renamed) program clause $B_0 \leftarrow B_1, \dots, B_n$ ($n \geq 0$) whose head unifies with A_i (resulting in an mgu, a substitution θ_1). If this is the case, the new goal clause G_1 will have the form $\leftarrow (A_1, \dots, A_{i-1}, B_1, \dots, B_n, A_{i+1}, \dots, A_m)\theta_1$. And as we have mentioned, the variables have been renamed now such that they are different from G_0 . Using the same resolution principle, one can derive a new goal clause G_2 from G_1 and so on. This process may either terminate or not. The case of not terminating is called an *infinite derivation*. There are two possibilities when it is not possible to derive G_{i+1} from G_i . One is when the selected subgoal cannot be resolved, i.e. it is not unifiable with the head of any program clause. This is called a *failed derivation*. The second case is when $G_i = \square$, i.e. the empty goal. This is called a *refutation*, and provides us with an answer to the original goal.

This process results in a finite or infinite sequence of goals starting with the initial goal. Each reasoning step is a pair $\langle G_i, C_i \rangle$, $i \geq 0$, where G_i is a goal and C_i is a program clause with renamed variables. The complete SLD-procedure is defined as follows:

DEFINITION 4.2.16 ▶ Let G_0 be a definite goal, P a definite program and S a selection rule. An **SLD-derivation** of G_0 (using P and S) is a finite or infinite sequence of goals:

$$G_0 \xrightarrow{C_0} G_1 \dots G_{n-1} \xrightarrow{C_{n-1}} G_n \dots$$

where each G_{i+1} is derived directly from G_i and a renamed program clause C_i via \mathcal{S} , following Equation 4.4.

Each SLD-derivation $G_0 \xrightarrow{C_0} G_1 \dots G_{n-1} \xrightarrow{C_{n-1}} G_n$ yields a sequence $\theta_1, \dots, \theta_n$ of mgus. The composition

$$\theta \equiv \begin{cases} \theta_1 \theta_2 \dots \theta_n & \text{if } n > 0 \\ \epsilon & \text{if } n = 0 \end{cases} \quad (4.5)$$

of mgus is called the *computed answer substitution* of the derivation. Note that there can be infinitely many derivations that are equivalent up to variable renaming. Note also that there can be multiple refutations of G_0 that compute the same answer substitution by a permutation of the sequence C_0, \dots, C_n of the (renamed) clauses in the derivation.

EXAMPLE 4.2.3 ▶ Let $p(X) \leftarrow q(X), r(X)$ be a clause and let KB consist of $q(a)$, $q(b)$ and $r(a)$. Let $\leftarrow p(Y)$ be a query. Applying resolution a couple of steps leads from $\leftarrow p(Y)$ to $\leftarrow q(Y), r(Y)$, to $\leftarrow q(a), r(a)$, to $\leftarrow r(a)$ and finally to \square , which means that the query $\leftarrow p(Y)$ succeeds with computed answer substitution $\theta = \{Y/a\}$. A second branch leads from $\leftarrow p(Y)$ to $\leftarrow q(Y), r(Y)$, to $\leftarrow q(b), r(b)$ to $\leftarrow r(b)$ and fails.

SLD-resolution is both sound and complete for definite programs. An SLD-derivation that ends in the empty goal corresponds to a refutation of (and provides answers to) the initial goal. All derivations together span a *tree*. The root of the tree is labeled G_0 , and if the tree contains a node labeled G_i and there is a renamed clause C_i in the program P such that G_{i+1} is derived from G_i and C_i then the node labeled G_i has a child node labeled G_{i+1} (and the edge connecting them is labeled C_i). The selection rule we have defined, builds the tree using a *depth-first* search strategy. A complete derivation tree is called a *proof tree*¹⁴. Backtracking is done on failed branches and on refutations (see e.g. Nilsson and Maluszinski, 1995). Another view at the clausal level is to see SLD-derivations as performing a *backward-chaining* procedure on the clauses. One useful aspect of SLD-resolution is its *goal-orientedness*; it automatically ensures (because it starts with the query) that it considers only those rules that are relevant for answering the query. Irrelevant rules are simply not considered in the course of the proof.

So far, SLD-resolution is limited to definite programs. However, when writing programs, it is natural to include *negative* conditions, i.e. to use general program clauses containing negative literals. The essence of logic programming is that there is an efficient *procedural semantics*, i.e. SLD-derivations. There is a natural way to adapt the procedural semantics to negation. The extension, called SLDNF (which stands for SLD with Negation as (finite) Failure) makes use of the *closed world assumption* (CWA) as a practical alteration to deal with negative literals. As an illustration, let P be the fragment of the program in Figure 4.6 which only contains the set of facts F plus the definition of *above/2*. The least Herbrand model $M(P)$ is $F \cup \{\text{above}(a, e), \text{above}(b, d), \text{above}(b, c), \text{above}(d, c)\}$. Note that neither P nor the Herbrand model include negative information such as d is *not* on top of b , or c is *not* on top of any block. This is quite natural in real-life or in relational databases. A database will represent the fact that my height is $1.94m$ but it will not represent explicitly that my height is *not* $1.61m$. In such cases, the lack of information will often be taken as

¹⁴And this has obvious connections to parsing and grammars.

evidence for the contrary. The CWA assumption is exactly this; a mechanism that allows us to draw negative conclusions based on the lack of positive information. Translated to the PROLOG case it states that unless an atomic sentence is known to be true, i.e. it can be derived, it can be assumed false.

The ideas of CWA and *negation as failure* (NAF) are used in the PROLOG language to reason with clauses and goals containing negative literals. In the following, we assume that logic programs consist of program clauses and we will sometimes explicitly refer to *general* logic programs and general goals. Consider the definition of `inTower1/1` in Figure 4.6. It contains one negative literal `not on(X, floor)`. The query `?- inTower1(a)` can be answered by first answering the query `?- on(a, floor)`. If this query succeeds, then the returned answer should be `no` and if the query fails, the answer should be `yes`.

The extension to SLDNF is done by making the following alteration. Let P be a general program and G_0 a general goal. If the selected literal of the current goal clause is positive, we proceed with the standard SLD procedure. If the selected literal is negative, i.e. it is `not A`, we make use of **i)** `not A` succeeds iff `?- A` has a finitely failed SLD-tree *with an empty answer substitution*, and **ii)** `not A` finitely fails iff `?- A` has an SLD-refutation. The extra condition in the first part needs some explanation. Consider the program P which consists of the clauses

$$\begin{array}{ll} \text{on}(a, b). & \text{onTop}(X) \leftarrow \text{not blocked}(X) \\ & \text{blocked}(X) \leftarrow \text{on}(Y, X) \end{array}$$

It should be clear that `onTop(a)` should be derived from the program when given the query `?- onTop(X)`. The problem however, is that `not blocked(X)` is called with X *uninstantiated*. Because the goal `on(Y, X)` has a refutation, with a non-empty answer substitution $\{Y/a, X/b\}$, the goal `onTop(X)` fails. Technically, SLDNF-resolution treats the subgoal `not blocked(X)` as $\forall X \neg \text{blocked}(X)$ instead of $\exists X \neg \text{blocked}(X)$. This problem is called *floundering* and shows that SLDNF-resolution cannot handle un-instantiated variables in negative goals correctly. This also implies that – as defined here and implemented in most PROLOG systems – SLDNF is essentially unsound. For most practical purposes however, this can be overcome by letting the programmer ensure that variables are in fact instantiated when calling a negative subgoal. Doing so makes negative atoms essentially a *test*.

Minimal Herbrand models do not immediately capture the intended semantics of programs under NAF. Consider the clause `p ← not q`. This program has two minimal models ($\{p\}$ and $\{q\}$), so there is no *unique* minimal model. Several solutions for giving proper semantics have been proposed so far (e.g. see Apt and Bol, 1994; Nilsson and Maluszinski, 1995), for example, *well-founded semantics*, *answer set programming* and *stratified programs*. Some of these solutions are more syntactic in nature (e.g. rewriting the program in some normal form) whereas others are more semantic. The *stable model semantics* defines a minimal model as a model where every atom in the model has a *justification*, i.e. a clause where the head is an atom and where every literal in the body is satisfied. In our example $\{p\}$ is a stable model, whereas $\{q\}$ is not.

NAF as implemented in PROLOG relies on a *procedural* mechanism of NAF. Technically speaking, it is unsound, and it relies on the programmer to ensure that entailments are in accordance with the intuitive meaning of the program. PROLOG provides the programmer with a number of *control* mechanisms to aid in this. One of the main methods is to make use of *orderings* and the use of the *cut* symbol. Clause orderings do not change the declarative meaning of a program, though they can provide some answers quicker (and

the ordering of atoms in a clause solve the floundering problem we described). For example, take the following program for computing the minimum of two values:

```
min(X, Y, X) : - X ≤ Y.
```

```
min(X, Y, Y) : - X > Y.
```

If we know that usually the first input is actually larger than the second, reversing the order of the clauses makes it more likely that the first clause gives us an answer. However, PROLOG will still try to get a refutation using the second rule too, even if we know that if the first clause provides us with a refutation, the second one will definitely not. A solution for this apparent inefficiency caused by the declarative interpretation is the *cut*-symbol (denoted '!'). Cuts can be used to *prune* the SLD(NF)-tree traversed, such that useless branches will not be considered. Take the following modification of the min-program¹⁵:

```
min(X, Y, X) : - X ≤ Y, !.
```

```
min(X, Y, Y).
```

The cut in the first rule forces that when the test $X \leq Y$ succeeds, the second clause will not be considered anymore, which renders the two clauses an if-then-else statement. In general the cut works as follows. Let $C \equiv H \leftarrow B_1, \dots, B_k, !, B_{k+2}, \dots, B_n$ be a program clause in a procedure defining H . If the current goal G unifies with the head of C , and B_1, \dots, B_k succeed, the cut has the following effects. The choice for C is committed to when reducing G and any other alternatives for A that might unify with G are ignored. Furthermore, if B_i fails (for $i > k + 1$) then backtracking happens only as far as the '!'. Any other choices remaining in the computation of B_i ($i \leq k$) are pruned from the tree. If backtracking finally reaches the cut, it fails and computation proceeds from the last choice made *before* the choice of G to reduce C . Cuts that do not change the declarative interpretation of the program, i.e. which only prune useless branches, are called *green cuts* whereas cuts that do are called *red cuts*.

PROLOG furthermore provides *structure inspection* (e.g. *pattern matching*), *lists*, *arithmetic*, and some *control structures* (e.g. *repeat-loops*). In addition *meta-logical* features are provided to modify the program itself (e.g. adding and retracting clauses) and for I/O-procedures. Many of these features trade declarativeness for computational advantages. An important extension of logic programming is *constraint logic programming* which is a very active area still, and we will see some aspects of that in Chapter 6.

4.2.2.3 OTHER FRAGMENTS AND EXTENSIONS OF FOL

There exist many variations on FOL differing in *expressiveness*, *computational properties* and *modeling capabilities*. Some logics are more powerful (such as higher-order logic) but many variations try to tradeoff the expressiveness with better computational properties, such as was the case in moving from FOL to Horn logic. Many *modal* logics enrich the semantic structures, and accordingly, extend the syntax with new operators. In this section we will briefly describe five main types of logics frequently used in the literature, and we

¹⁵Note that this program is still not completely correct. Take as goal $\leftarrow \text{min}(2, 4, 4)$. This goal succeeds, implying that 4 is the minimum of 2 and 3, which is obviously not correct. The reason for this is that the third argument was intended as an *output* variable, but now used as an *input* (i.e. it is instantiated). The corrected program is

```
min(X, Y, X) : - X ≤ Y, !.
```

```
min(X, Y, Y) : - X ≥ Y.
```

which will work correctly for all cases. This program is true in the intended model and now $\leftarrow \text{min}(2, 4, 4)$ does not succeed anymore.

highlight some connections with relational RL.

Typed Logic. For some systems, it can be convenient to distinguish between different *types* (or, *sorts*) of objects. A *sorted* first-order language has variables and terms of different sorts. Semantically this means that the domain is partitioned into (disjoint) sub-domains, one for each sort. Variables and terms will denote elements in their corresponding sub-domains. In BLOCKS WORLD one might introduce types *block* and *surface*, such that the *type definition* of *on/2* is $block \times \{block, surface\}$, i.e. a block can be on some other block or a surface, but never can a surface be on a block. Quantification will range over sub-domains only, and each predicate symbol will be restricted to taking only arguments from specific types.

It turns out that many-sorted FOL can always be reduced to standard logic, such that the former is not more powerful than the latter. However, for various reasons, other than modeling convenience and comprehensibility, types can be useful for reasoning and manipulation purposes. For example, when types are used, the number of substitutions is reduced because sub-domains have less elements. This speeds up inference steps such as in resolution (see Equation 4.4). Furthermore, types are very useful in *inductive* logic (see Section 4.3), because they reduce the *search space*. Types can always be emulated by introducing unary relation symbols such as *block/1*. However, these emulated types create additional reasoning steps, because they introduce extra atoms and substitutions. Most (practical) systems using types deal with them on a meta-level. Many inductive logic systems and action logics (see further sections) support the use of types.

Description Logics. *Description logics* (DL) are notations that make it easier to represent *definitions* and *properties of categories*, slightly in contrast to FOL that is focused on describing objects. DL are a limited form of FOL, but in return provide better computational properties, e.g. DL usually do not provide *negation* and *disjunction*. The main rule of inference used is *subsumption* (see next sections). As an example, the FOL formula $block(X) \leftrightarrow square(X) \wedge solid(X) \wedge wood(X)$ can be represented as a *concept* in DL as $block = AND(square, solid, wood)$. An advantage of reasoning in DL is the fact that usually quantification is implicit, i.e. there is no variable in the second definition.

In relational RL, the API system by Fern *et al.* (2006) uses a *taxonomic syntax* as their base representation of policies for relational RL. Brachman and Levesque (2004, Chap. 9) and Russell and Norvig (2003, Sec. 10.6) discuss DL and taxonomic syntax in more detail.

Higher-Order Logic. FOL is called *first-order* because it can quantify over first-class citizens: objects. However, consider the following example. Let $\{red/1, blue/1, yellow/1\}$ be unary predicates denoting block colors, e.g. $red(a)$ says that block *a* is red. *Higher-order logics* (HOL) (see Lloyd, 2003) are logics where one can quantify *over relations* as well. For example, the formula $\forall Color \forall B (Color(B) \wedge ((Color = red) \vee (Color = blue)) \rightarrow nice(B))$ is an example in *second-order logic*, which quantifies over colors and says that a block is *nice* if it is red or blue. In contrast to many other extensions of FOL, HOL is strictly more powerful than FOL, because it can express sentences that cannot be expressed finitely in FOL. A good example is the *transitive closure* of a relation *P*.

The use of HOL in the relational RL setting was investigated by Cole *et al.* (2003) based on the ALKEMY HOL tree learning implementation (see Lloyd, 2003).

Modal Logic. First-order *Modal logic* (or, *intensional logic*) extends the FOL syntax with *modal operators* that take sentences as arguments. Typical examples are $\Box\varphi$ (φ is *definitely true*), $\Diamond\varphi$ (φ *might be true*). *Epistemic logic* (Fagin *et al.*, 1996) deals with *modalities* such as *believes* ($B\varphi$) and *knows* ($K\varphi$), whereas *temporal logics* often use notions such as φ *will always be true* and φ *will eventually be true*. Semantics for these modalities is usually given by *possible worlds semantics* (Kripke, 1963). Let Γ be a first-order language and let W be the set of Γ -structures, i.e. the *set of possible worlds*. Furthermore, let R be an *accessibility relation* on the set W . Semantics for formulas without modalities are equivalent to Definition 4.2.7, with an added index $w \in W$ (e.g. $(\mathcal{A}, \mathcal{V}, w) \models \varphi$). Now the *might* (i.e. \Diamond) modality is interpreted as *there should be some world accessible from the current world in which the formula holds*, i.e. $(\mathcal{A}, \mathcal{V}, w) \models \varphi$ iff $\exists w' \in W(R(w, w') \wedge (\mathcal{A}, \mathcal{V}, w') \models \varphi)$. Analogously, $\Box\varphi$ requires that φ is true in *all worlds accessible*, i.e. $\forall w' \in W(R(w, w') \wedge (\mathcal{A}, \mathcal{V}, w') \models \varphi)$. In epistemic logic, the accessible worlds are usually interpreted as the *belief state* which contains all worlds that could be the current, real, world. In temporal logics, this set can contain all worlds that are possible future belief states. Many different axiom systems exist, all giving different semantics to the modalities. Most, however, render the accessibility relation an equivalence relation.

Combining modalities and FOL introduces many (philosophical and technical) difficulties with nesting and quantification (see Gamut, 1991, for a good description). Many action formalisms (of which we treat some basics in Section 4.4) have been extended to deal with epistemic modalities and belief states (see also Wooldridge, 2002). Explicit use of modal logics has so far not yet been investigated in the relational RL setting. However, there are obvious relations with the POMDP (see Section 2.7) and SMDP (see Chapter 3) settings when looking at possible worlds semantics, and epistemic and temporal logics. Furthermore, scaling relational RL to the level of cognitive agents that have rich belief structures will involve understanding how epistemic and temporal logic can be combined with RL (see Chapter 7 for more on these matters).

Probabilistic Logics. A last type of logic we consider, extends the meaning of truth from the standard 2-value setting (i.e. true and false, or 1 and 0) to *probabilities* between 0 and 1. Combining logic and probability is a long-standing problem in mathematical logic and philosophy (see Galavotti, 2005, for an extensive discussion). However, from a (technical) AI point of view, it is becoming increasingly understood how to build effective representations and algorithms to deal with probability distributions over first-order structures (see Halpern, 2003). Probabilistic propositional logic shares many characteristics with first-order versions (e.g. probability distributions over sets of models, see Nilsson, 1986; Halpern, 2003). In Section 4.3.3 we will discuss recent combinations of probability and logical *induction*, a direction that is getting increasingly more mature.

Probabilistic FOL (PFOL) was independently approached by both Bacchus (1990) and Halpern (1990). The basic solution is to introduce modal operators for statements such as $P(\varphi) = 0.7$ meaning that the probability that φ holds is 0.7, given the current belief state. In contrast to probabilistic propositional logic, due to a separate *domain* of individuals in PFOL two fundamental types of statements arise. Let us stick to the standard *bird* example used in many papers. Consider the formula $P(\forall X(\text{bird}(X) \rightarrow \text{flies}(X))) > 0.9$. This statement talks about statistics in some domain and can be interpreted as *at least 90 percent of all birds in the domain flies*. Another example is $\varphi \equiv P(\text{flies}(\text{tweety})) > 0.9$. In this case, the problem is that in the current world, *tweety* either flies or not (thus

contradicting the > 0.9). It is therefore that this statement is different from the first and it is called a *degree of belief*. The semantics is given in a similar way as for *belief* in modal logics; φ is true if *tweety* flies in at least 90 percent of all *accessible worlds*. Many recent, efficient ML systems have appeared that are based on these logics (see Section 4.3.3), but most are aimed at modeling degrees of belief.

The use of PFOL for relational RL has been limited so far, because most approaches focus on fully observable RMDPs (so, no probabilistic belief states are necessary) and the only probabilistic representations are given by (stochastic) action languages. Examples of the use of efficient PFOL-based formalisms are given by Guestrin *et al.* (2003a) and Croonenborghs *et al.* (2006a).

4.2.3 First-Order Abstraction and Generalization

Declarativeness is an important aspect of logic-based formalisms. The aim of declarativeness is to hide the selection mechanism (e.g. how and when to use specific parts of the knowledge base) from the programmer. However, as we have already seen in the PROLOG language, moving away from that somewhat and employing additional procedural semantics, offers significant computational advantages. Dietterich (2003) discusses this point in full detail, and addresses the aspect of *reasoning* in ML in general. Here we have a restricted view in mind. One of the main purposes of logic in ML, action formalisms and relational RL is efficiently representing *sets* of interpretations that denote e.g. *concepts*, *classes*, *states* and *actions*. As in many abstraction methods for MDPs (see Chapter 3), these abstractions are typically *partitions*, defining sets of states that have the same value, have the same (optimal) action, or have the same transition characteristics. FOL is powerful enough to describe partitions using complex formulas. However, this power is often unnecessary when representations such as *trees* and *decision lists* are used.

4.2.3.1 EFFICIENT REPRESENTATIONS WITH PROCEDURAL SEMANTICS

A number of efficient representations for first-order abstraction keeps entailment as the principle method for abstraction (e.g. over interpretations) but makes use of the *redundancy* in the description of the partitions. Deciding entailment requires a procedural semantics of the abstractions, for example an ordering on a set of formulas.

Decision Lists. Decision lists have been lifted to the first-order case (Mooney and Califf, 1995). Many systems in this book use a list of clauses $\{H_i \leftarrow B_i\}$. Often the head H_i can be interpreted as a *class value* and the body B_i is interpreted as stating *conditions* on the current input (i.e. an interpretation). The first rule $H_j \leftarrow B_j$ for which $i \models B_j$ with answer substitution θ returns $H_j\theta$. For example, consider the following *abstract* policy for a BLOCKS WORLD. It optimally encodes the policy for reaching states where $\text{on}(a, b)$ holds:

```

r1 : move(X, floor) ← onTop(X, b)
r2 : move(Y, floor) ← onTop(Y, a)
r3 : move(a, b)     ← clear(a), clear(b)
r4 : noop          ← on(a, b)

```

where *noop* denotes doing nothing. The rules should be read from top to bottom. Given a state, the first rule where the abstract state applies, generates an optimal action. A PROLOG implementation of this decision list includes a cut symbol at the end of each clause. Many of the model-free (see Chapter 5) and model-based (see Chapter 6) approaches in rela-

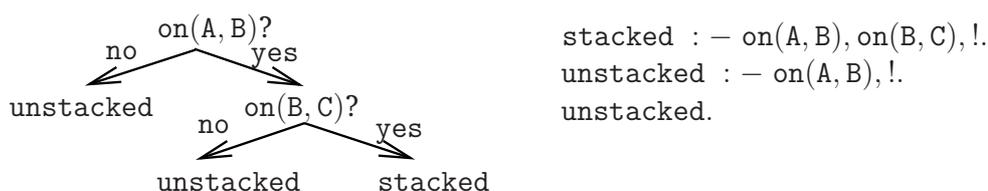


Figure 4.7: a) A FOL *decision tree*. b) A PROLOG *decision list*.

tional RL use decision lists for abstraction. The first application for model-free relational RL was described by Lecoecue (2001) whereas for model-based by REBEL (Kersting *et al.*, 2004; van Otterlo *et al.*, 2004). Earlier Khardon (1999a) and Martin and Geffner (2000, 2004) employed decision lists (or, *production rules strategies*) for learning policies in a supervised setting.

Explicit Partitions. Decision lists make use of the *order* imposed on the rules. If *negation* is part of the logical language used, *explicit* partitions can be defined. For example, if we have as a first rule φ , then a second rule can be denoted $\neg\varphi \wedge \psi$. In the example above, we could define a partition starting with

$r_3 : \text{move}(a, b) \leftarrow \text{clear}(a), \text{clear}(b)$

and add the following rule denoting what to do if a and b are not both clear¹⁶:

$r_5 : \text{move}(X, \text{floor}) \leftarrow \text{not}(\text{clear}(a), \text{clear}(b)), \text{onTop}(X, b)$

Relational RL systems such as SDP and FOALP (see Chapter 6) use *case* statements consisting of complex logical formulas defining explicit partitions, though most other systems lack this expressivity.

First-Order Trees. Logical decision trees (see De Raedt *et al.*, 2001; Kramer and Widmer, 2001) are efficient representations of partitions. A well-known system for logical classification (Blokceel and De Raedt, 1998) and regression trees (Karalic and Bratko, 1997; Blokceel *et al.*, 1998) is TILDE (Blokceel, 1998). Figure 4.7 shows an example in which the input space is partitioned using logical tests in the nodes in the tree such as $\text{on}(A, B)?$. An important difference with propositional tree representations is that nodes can share variables. When given an input i , first the root node atom $\text{on}(A, B)?$ is tested against the example. If the test succeeds, producing substitution θ , one follows the *yes*-branch and applies the following test (substituting variables according to θ). If the test fails, one follows the *no*-branch. This process continues until finally arriving at a *leaf* node, which contains the classification (e.g. *stacked* and *unstacked* in this example) for this input. Leaf nodes of first-order *regression* trees contain real values, instead of symbolic class values.

There is a unique path leading to each leaf node, which implies that the tree represents a partition¹⁷. Logical decision trees can be converted into a decision list, i.e. by employing a set of rules with the use of cuts. The tree in Figure 4.7a has the same semantics as the decision list in Figure 4.7b. A tree representation of the decision list in the example above can differ in the order of the tests, based on statistics of occurrence, e.g. states in which $\text{clear}(a), \text{clear}(b)$ holds are less frequent, so an optimized tree will test for this property lower in the tree. *Higher-order* logic tree representations are similar to first-order

¹⁶Note that in this example, the $\text{onTop}(X, Y)$ relation is defined in terms of the fact that Y is not clear, for at least the block X is above Y .

¹⁷There can be many different substitutions though.

representations, though the tests in the trees can be much more complex (see Lloyd, 2003).

In relational RL, the first system that was introduced used the logical regression tree learner TILDE-RT (Blockeel *et al.*, 1998) that was later improved to work in an *incremental* fashion (Driessens *et al.*, 2001a). Higher-order logical trees were used in the relational RL setting by (Cole *et al.*, 2003) using the tree learner ALKEMY. Recently, Croonenborghs *et al.* (2007b) use first-order *probability trees* (i.e. inducing probability distributions over atoms) as partial representation of the transition function for a relational MDP

First-Order Decision Diagrams. In the previous chapter we have described BDDs and ADDs (See Section 3.5), in the context of the SPUDD and APRICODD algorithms for factored MDP representations. Propositional BDDs have been lifted to the first-order case (Groote and Tveretina, 2003). The use of variables, and the sharing of them among nodes in the diagram enable a similar kind of compactness of the representation as with first-order trees. Joshi *et al.* (2006); Wang *et al.* (2007) extend the work by Groote and Tveretina (2003) to ADDs with numerical leaves through the use of an aggregation function. Numerical leaves enable the first-order ADDs to represent first-order *value functions* and *transition functions*. As in the propositional case, first-order BDDs and ADDs come with a number of efficient operations and transformations on the diagrams – such as reductions, simplifications and combinations – that help to keep their size small. In addition, background knowledge can be used in the reductions.

4.2.3.2 DISTANCES, KERNELS AND FEATURES

Instead of keeping the standard syntax–semantics dichotomy and trying to efficiently represent *partitions* by means of logical abstractions, some other, more recent methods, try to span a new *generalization space* over the set of relational structures. Remember from the propositional setting that each example can naturally be represented in an Euclidean space in which a default distance measure is defined. The structure of this space is used by many learning algorithms to form concepts and clusters. For relational examples, there is no such natural space and in order to define it, people have looked into three directions. One aspect these methods share, is that they transform the learning problem into a new space. The main advantage is that standard (propositional) algorithms apply in this new space. A downside is that *comprehensibility* is often considerably decreased.

The first direction is to define *distances* between interpretations. Ramon (2002) extends the concept of a distance to *strings*, *sets* and *interpretations*, among others. Once such a distance measure is in place, standard clustering algorithms apply. Driessens and Ramon (2003) applied this idea in relational RL. Relational distances are computationally expensive, though they allow for the application of existing (propositional) learning algorithms (see also Kirsten *et al.*, 2001).

The second direction is to define first-order *features*. A set of first-order features $\varphi_1, \dots, \varphi_n$ is a set of first-order formulas that act as *binary features* where for some current input i (i.e. an interpretation) the value of a feature φ is 1 if φ covers i and 0 otherwise. The resulting binary feature vector $\{0, 1\}^n$ can be used in a weighted combination, such as in linear VFA (see Section 3.6). First-order features have been used e.g in model-free RL by Walker *et al.* (2004) and in model-based methods by Sanner and Boutilier (2006).

The third direction, related to the idea of first-order distances is the definition of *kernels* for relational interpretations. Working with kernels amounts to learning in an *inner product*

space, which can be extremely large for most applications but can be used for learning due to the so-called *kernel trick* (Schölkopf and Smola, 2002). There are some applications of kernel-based methods in the field of relational RL (Gärtner *et al.*, 2003; Walker *et al.*, 2004; Driessens *et al.*, 2006b; Asgharbeygi *et al.*, 2006). Some other systems such as RIB and KBR (see Section 5.3.2) define abstraction levels by inducing high-level (first-order) features, or by using *kernels* and *distances* on relational states. Generalization and abstraction is then performed in the new high-dimensional space spanned. These allow for advanced function approximators to be used, although the induced abstraction levels lose *comprehensibility*.

An additional feature of first-order languages for abstraction is the possibility of using *background knowledge*, thereby extending the language used for abstraction. The `onTop/2` relation used in the examples can be defined by the user (in terms of `on/2` and `clear/1`) and used to create powerful, compact abstractions. First-order background knowledge can provide a natural means to enhance the language for abstraction levels, and can be used in various induction and generalization algorithms.

4.3. Learning in First-Order Domains

First-order ML deals with the problem of how to learn logical abstractions from data. In the previous section we have discussed how to *represent* and *model* logical abstractions, and in addition, how to *reason* with these logical abstractions. In the current section we outline *inductive logic* and ML techniques that can cope with logical representations of concepts. Analogous to the previous section, we will see that – although full FOL would be desirable – most induction techniques are applied in the restricted context of clausal logic for obvious reasons of computational complexity. Indeed, ILP methods (see Section 4.3.2) focus on the induction of various sorts of logic program representations.

Broadly speaking, there are three approaches to the induction of logical abstractions. First, in Section 4.3.1 we explain what induction means in a semantic way. Second, in Section 4.3.2 we outline some of the core techniques of the established field of *inductive logic programming* (ILP) at a level that is sufficient for understanding the rest of this book. ILP has introduced a large number of approaches and systems to learn clausal theories from data. Third, ILP has recently been extended to *statistical relational learning* (SRL), which will be the topic of Section 4.3.3. SRL has primarily focused on two important problems. One is the combination of logic and probability, similar in spirit to the probabilistic logics in Section 4.2.2.3, but with a strong emphasis on computational efficiency and learnability. The second focus is on dealing with various kinds of uncertainty present in most common ML tasks. This has resulted in many standard ML algorithms to be *lifted* (see van Laer and De Raedt, 2001b,a) to the first-order case, resulting in e.g. relational Bayesian networks, Markov networks and decision trees.

4.3.1 Obtaining Logical Abstractions

In the previous section we described how to reason in FOL, assuming a theory, questions and axioms are given a priori. It is desirable to obtain these structures automatically, either from data, or from existing knowledge. For example, if it turns out that for all states visited that required the action `left` there was some blue object on top of some green object, the system could automatically induce a rule such as $\exists X, Y(\text{on}(X, Y) \wedge \text{blue}(X) \wedge \text{green}(Y) \wedge X \neq$

$Y) \rightarrow \text{left}$. Relational RL introduces the same kind of opportunities for automatically learning abstractions as the methods described in Chapter 3. The real difference is the use of FOL, and with that, new induction methods that are based on FOL.

An induction problem is to find an abstraction level compactly abstracting some aspect of the target domain. It has the following characteristics. First, there is a *signature of the target function*, that specifies the *form* of the structures we want to learn. Second, a *coverage function* determines when an example is covered. Third, a set of *examples* is a (presumably) representative selection of models that the final induced function should cover. Fourth, a *hypothesis language* determines the syntactic elements that can be used to form expressions. This language also determines the *search space* in which the target function should be found. Finally, a *background theory* can be provided which may restrict the set of possible target functions. For example, one may possess a priori knowledge about a domain such that the induction process can be more focused, by making use of existing knowledge. A main goal of the induction process is to find a target function that *generalizes* correctly over unseen examples.

Semantically, the difference between deduction and induction can be characterized precisely (Denecker *et al.*, 1996)¹⁸. Let \mathcal{K} be a possible worlds (e.g. a set of first-order structures) model for a given logical language. The process we have described as *deduction*, with a concrete example implementation given by SLDNF-derivations, can be formalized as follows. Logical *deduction* is to verify – given a possible worlds model \mathcal{K} and a formula F – whether $\forall M \in \mathcal{K} : M \models F$. Deduction is a core process for reasoning in FOL, and we can equate it with other reasoning styles such as *theorem proving*, *planning* and many more. In the *induction* setting an entire class Γ of possible worlds models is given. Every $\mathcal{K} \in \Gamma$ represents one possible combination of general laws that can hold in the actual, real world. A set of examples E renders concrete scenario knowledge. The goal of induction can then be understood as finding a $\mathcal{K} \in \Gamma$ that *explains* the examples. There are a number of methods, but one is *learning from interpretations*: find a $\mathcal{K} \in \Gamma$ such that $\forall e \in E : e \in \mathcal{K}$ (resp. $e \notin \mathcal{K}$ for a negative example e). Notice that in all cases, *induction refines* Γ . In some sense induction can be considered as the opposite of deduction¹⁹. Where in deduction, we reason from the perspective of a formula F and we want to assess its validity, in inductive reasoning, we reason from a concrete set of examples and we want to find a possible worlds model that is consistent with our examples.

The main learning setting studied in ML and in fact in this book, is the inductive setting. Still, many other settings exist that are based on *deduction*, *abduction*, *explanation-based generalization* and prior knowledge. These are known under the general name *analytical learning* (Mitchell, 1997, Chap. 11) and they use reasoning (i.e. deduction) to generate new knowledge from prior knowledge. Many of the algorithms discussed in this book use various deductive methods to generate new structures and we will discuss them when needed. One of these analytical methods is *explanation-based learning* (EBL). We have encountered EBL briefly in Section 3.5.2 and we will deal more extensively with this type of learning in Chapter 6. There, we discuss model-based, relational RL techniques that all have their roots in explanation-based generalization.

¹⁸Note that this use of possible worlds model is different from – though related to – the possible worlds semantics described in Section 4.2.2.3.

¹⁹Note that this is an overstatement. On an intuitive level it holds, but the technical part is much more complex and subtle.

4.3.2 Inductive Logic Programming

The field of *inductive logic programming* (ILP) (Muggleton and De Raedt, 1994; Lavrac and Džeroski, 1994; Flach, 1994; Bergadano and Gunetti, 1995; Nienhuys-Cheng and de Wolf, 1997; Džeroski and Lavrac, 2001b; Džeroski, 2003) has developed many methods to learn logical abstractions. ILP resides on the cross section between logic programming and inductive learning. Its roots lie in *inductive concept learning* which has been studied extensively, and where the aim is to discover a set of classification rules – given a pre-classified set of examples – that has high predictive power and can be used to classify unclassified examples. ILP overcomes two limitations of classical, propositional inductive concept learning: **i)** the use of relatively simple representation languages (essentially propositional, see also Section 4.1.3), and **ii)** the difficulties of using substantial background knowledge in the learning process. Logic offers an elegant mechanism to incorporate prior knowledge about the domain into the induction process. The emphasis on one universal representation language for e.g. concepts, examples, background knowledge, declarative bias, in addition to a close connection between syntax and semantics (i.e. Herbrand semantics, see Section 4.2.2.1), gives ILP a wide scope of applicability.

ILP concept definitions are represented as clausal theories. The theory that is obtained by the learning algorithm is dependent on the particular *language bias* (i.e. the language in which the theory can be expressed). Furthermore, the theory is supposed to *cover* all positive examples, and none of the negative examples. The notion of *coverage* depends on the learning setting and the particular *form* examples can take, and these aspects will be made more concrete later in this section. At this point, we can formulate a general definition of ILP problems:

DEFINITION 4.3.1 ► The basic form of an **ILP problem** is defined as follows.

Given: a set \mathbb{H} of possible programs,
 a set $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$ of positive and negative examples,
 a consistent (definite) background theory \mathcal{B}
 a covers relation $cv(\mathcal{H}, \mathcal{B}, e)$

Find: a logic program $\mathcal{H} \in \mathbb{H}$ that is complete and consistent, i.e. such that:
 \mathcal{B} should not cover any $e \in \mathcal{E}^-$ (prior satisfiability)
 $\mathcal{H} \cup \mathcal{B}$ should not cover any $e \in \mathcal{E}^-$ (posterior satisfiability)
 Some $e \in \mathcal{E}^+$ are not covered by \mathcal{B} (prior necessity)
 All $e \in \mathcal{E}^+$ are covered by $\mathcal{H} \cup \mathcal{B}$ (posterior sufficiency)

Here, the program \mathcal{B} consists of the *background knowledge* that is possibly available. The program $\mathcal{B} \cup \mathcal{H}$ consists of all clauses occurring in \mathcal{B} and \mathcal{H} . The set of programs \mathbb{H} is called the *hypothesis space* (or *search space*). The desired result is a logic program (i.e. a set of clauses) that – together with the available background knowledge – covers the (positive) examples. Note that if the data contains noise, or is probabilistic (see further Section 4.3.3) the last two criteria do not have to be satisfied completely.

Consider the data and (extensive) background theory depicted in Figure 4.8. There are two positive (\oplus) and two negative (\ominus) examples of the relation *daughter/2*. In the language of Horn clauses, a (learned) hypothesis could be the clause $\text{daughter}(X, Y) \leftarrow \text{female}(X), \text{parent}(Y, X)$. Depending on the availability of other background clauses²⁰ such

²⁰These definitions are very simple: $\text{parent}(X, Y) : - \text{father}(X, Y)$. and $\text{parent}(X, Y) : - \text{mother}(X, Y)$.

Training examples		Background knowledge
daughter(jose, greet)	\oplus	parent(greet, jose). female(jose).
daughter(marieke, carel)	\oplus	parent(wim, jose). female(greet).
daughter(wim, jose)	\ominus	parent(carel, marieke). female(marieke).
daughter(marieke, martijn)	\ominus	parent(greet, martijn). female(rieky).

Figure 4.8: Data and background knowledge for learning the daughter relation.

as $\text{mother}/2$ or $\text{father}(Y, X)$ the induced hypothesis could take the alternative form:

$$\begin{aligned} \text{daughter}(X, Y) &\leftarrow \text{female}(X), \text{father}(Y, X). \\ \text{daughter}(X, Y) &\leftarrow \text{female}(X), \text{mother}(Y, X). \end{aligned}$$

This hypothesis is consistent and complete w.r.t. the background theory once extended with definitions for $\text{mother}/2$ and $\text{father}/2$.

4.3.2.1 SETTINGS

Within the field of ILP, several variations of Definition 4.3.1 exist that differ in the way examples are modeled, and in how coverage is defined. The standard setting in ILP is the *learning from entailment* setting. Typical examples are FOIL (Quinlan, 1990) and PROGOL (Muggleton, 1995), and many of the earlier systems in ILP have investigated this setting (see for overviews Muggleton and De Raedt, 1994; Lavrac and Džeroski, 1994; Flach, 1994; Bergadano and Gunetti, 1995).

DEFINITION 4.3.2 ► The **learning from entailment** setting is defined as follows. The example set consists of definite clauses, in most systems restricted to ground facts. A hypothesis \mathcal{H} **covers** an example e in this setting w.r.t. a background theory \mathcal{B} , denoted $\text{cv}_{\text{ent}}(\mathcal{H} \cup \mathcal{B}, e)$, iff $\mathcal{H} \cup \mathcal{B} \models e$. Consequently, the function cv_{ent} is defined as $\text{cv}_{\text{ent}}(\mathcal{H} \cup \mathcal{B}, \mathcal{E}) = \{e \in \mathcal{E} \mid \mathcal{B} \cup \mathcal{H} \models e\}$.

Thus, each example is usually a fact (i.e. a special case of definite clauses) and the resulting hypothesis \mathcal{H} (e.g. a definite program) should entail all positive examples, but none of the negative ones. The example in Figure 4.8 belongs to this setting.

A second setting, more recent and particularly relevant for the work in relational RL and indeed this book, is the *learning from interpretations* setting (see De Raedt, 1997; Blockeel *et al.*, 2000a; van Laer, 2002).

DEFINITION 4.3.3 ► The **learning from interpretations** setting is defined as follows. The example set consists of Herbrand interpretations, which completely describe the examples. A hypothesis \mathcal{H} **covers** an example e in this setting, w.r.t. a background theory \mathcal{B} , denoted $\text{cv}_{\text{int}}(\mathcal{H} \cup \mathcal{B}, e)$, iff e is a model of $\mathcal{H} \cup \mathcal{B}$, i.e. $e \models \mathcal{H} \cup \mathcal{B}$. Consequently, the function cv_{int} is defined as $\text{cv}_{\text{int}}(\mathcal{H} \cup \mathcal{B}, \mathcal{E}) = \{e \in \mathcal{E} \mid e \models \mathcal{B} \cup \mathcal{H}\}$.

This setting is much closer to propositional and attribute-value representations than the learning from entailment setting. Each example is a complete picture of the world, i.e. a Herbrand interpretation. Typical examples of systems using this setting are TILDE (Blockeel and De Raedt, 1998; Blockeel *et al.*, 1998; Blockeel, 1998) and CLAUDIEN (De Raedt and Dehaspe, 1997). In fact, algorithms in this settings can be seen as first-order *upgrades*

of traditional, propositional ML algorithms (see van Laer, 2002, who defines a methodology²¹ around this aspect).

The key difference between learning from entailment and learning from interpretations is that in the latter examples carry much more information. This makes learning more tractable in this setting (De Raedt, 1997). In part, this is due to the fact that in the learning from interpretations setting, coverage of examples can be tested *locally*, independently of the other examples (Blockeel *et al.*, 2000a). In the entailment setting, the data consists of multiple tuples from multiple relations. For testing coverage, one needs a database system to solve complex queries (or a theorem prover). The interpretations setting is closer to attribute-value representations in this respect too. Systems that use local coverage tests are much easier to scale up than those using global tests.

Learning from interpretations and learning from entailment is largely done in the same way. The main difference lies within the notion of generality (see more below). In both settings it holds that a hypothesis \mathcal{H}_1 is *more general than* another hypothesis \mathcal{H}_2 if all the examples covered by \mathcal{H}_2 are also covered by \mathcal{H}_1 . Based on the definitions of cv_{ent} and cv_{int} we see that in the entailment setting, \mathcal{H}_1 is more general than \mathcal{H}_2 iff $\mathcal{H}_1 \models \mathcal{H}_2$ whereas in the interpretation setting iff $\mathcal{H}_2 \models \mathcal{H}_1$.

In addition to the settings defined in Definitions 4.3.2 and 4.3.3, a third setting is the *learning from proofs* setting (see De Raedt and Kersting, 2004), in which the examples are complete proof trees and coverage equals an example being a proof tree for the hypothesis w.r.t. the background theory. Applications in which data consists of proofs can be found in natural language processing (e.g. *parse trees*) and other grammar-driven data (e.g. structured markup languages). A natural induction method would be bottom-up generalization from traces. Furthermore, this setting might be applied for model learning in relational RL, i.e. learning from action execution traces. A fourth setting is a generalization of learning from (partial) interpretations, called *learning from satisfiability*, and it allows for examples to be full clausal theories. Let \mathcal{H} and e be clausal theories. Then, \mathcal{H} covers e under satisfiability iff $\mathcal{H} \cup e \not\models \square$. Both learning from entailment and learning from interpretations can be seen as special cases of this setting (De Raedt, 1997).

4.3.2.2 STRUCTURING THE SPACE OF CLAUSES

Regardless of the specific setting used in the learning process, in order to search efficiently in the space of hypotheses (see Definition 4.3.1), one needs a *generality order* such that hypotheses can be compared, the search can be guided, and irrelevant parts of the space pruned. Various generality frameworks have been proposed, including *θ -subsumption* (Plotkin, 1970) and *inverse implication, resolution and entailment* (see Nienhuys-Cheng and de Wolf, 1997, for an extensive overview). The large majority of ILP systems uses *θ -subsumption*, for reasons of efficiency (i.e. it is decidable – though NP-complete – where most others are not) and simplicity (the idea is well-understood and deterministic, in contrast to some other methods). A disadvantage of *θ -subsumption* is that it works on the level of individual clauses only, such that it creates difficulties when learning *recursive* clauses. Additionally, some extensions are needed to use it in the context of a background theory. Nevertheless, in the remainder of this book we will be mainly concerned with *θ -subsumption* as a generality framework. Let us first define *θ -subsumption* for clauses.

²¹Also known as "the Leuven methodology", after the academic hometown of that author.

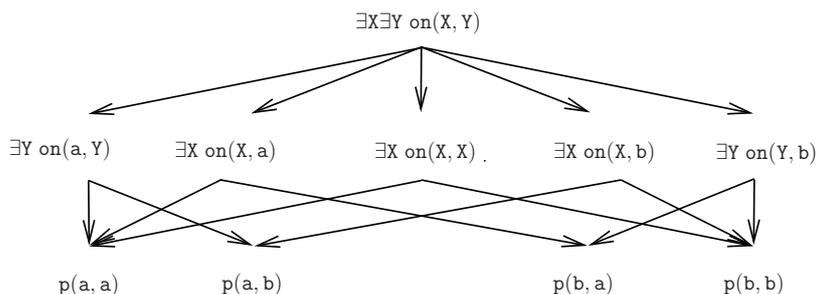


Figure 4.9: Lattice (of queries) restricted to one literal, based on the generality ordering \preceq_θ .

Note that in the following definition, we use a set notation of clauses.

DEFINITION 4.3.4 ▶ Let C and C' be two program clauses. Clause C θ -**subsumes** C' , denoted $C' \preceq_\theta C$, if there exists a substitution θ such that $C\theta \subseteq C'$.

As an example, let C_1 be the clause $a(X) : \neg b(X)$ and let C_2 be $a(c) : \neg b(c), b(d)$. Now, C_1 θ -subsumes C_2 with the substitution $\{X/c\}$. θ -subsumption introduces a syntactic notion of *generality*. A clause C_1 is *at least as general as* clause C_2 ($C_2 \preceq_\theta C_1$), if C_1 θ -subsumes C_2 . Clause C_1 is *more general than* C_2 ($C_2 \prec_\theta C_1$) if $C_1 \preceq_\theta C_2$ holds and $C_2 \preceq_\theta C_1$ not. θ -subsumption is reflexive and transitive, but not anti-symmetric (e.g. $a(X) : \neg b(X)$ and $a(X) : \neg b(X), b(Y)$ θ -subsume each other). θ -subsumption induces a *partial order* on the space of clauses, i.e. it induces a *lattice*. Figure 4.9 shows a lattice for one literal ($\text{on}/2$) and two constants (a and b). A *reduced clause* is a clause that does not θ -subsume any of its sub-clauses. (see Gottlob and Fermüller, 1993; Nienhuys-Cheng and de Wolf, 1997, and also Chapter 6). Reducing a clause is semi-linear in the number of literals (it makes a linear number of calls to a θ -subsumption check, which itself is an NP-complete problem). Every equivalence class contains a reduced clause that is unique up to variable renaming. The fact that these equivalence classes form a lattice entails that every two clauses have a unique *upper* and *lower bound* under θ -subsumption.

One important property of θ -subsumption is that it is close to logical entailment. If clause C_1 θ -subsumes clause C_2 , then C_1 logically entails C_2 , i.e. $C_1 \models C_2$. The reverse is not always the case. There are many relations between θ -subsumption and entailment, but in general it holds that C_1 θ -subsumes C_2 *without entailing it* if the resolution proof of C_2 from C_1 requires to use C_1 more than once. This may be the case, for example, for recursive clauses and some functor occurrences.

θ -subsumption is a purely *syntactic* notion of generality. Although the entailment (\models) ordering would be preferred – for reasons of generality – there are several problems. Calculating the least upper bound of two clauses would be given by the disjunction of the two clauses, though this may not be a Horn clause. Furthermore, generalizations and specializations of clauses are not easily computed under entailment, whereas for θ -subsumption these are relatively easy, syntactic operations. Finally, entailment between clauses is *undecidable*, whereas θ -subsumption is decidable (but NP-complete). Furthermore, a number of efficient implementations of θ -subsumption have been studied (e.g. see Kietz and Lübke, 1994; Skvortsova, 2006b).

Deciding θ -subsumption in the context of a background theory is slightly more complicated. Several methods have been proposed, among which Plotkin (1970)'s *relative*

Algorithm 7 A generic ILP algorithm (Lavrac and Džeroski, 1994)

```

1: Initialize  $\mathcal{E}_{cur} := \mathcal{E}$ 
2: Initialize  $\mathcal{H} := \emptyset$ 
3: repeat
4:   % COVERING
5:   Initialize  $c := T \leftarrow$ 
6:   repeat
7:     % SPECIALIZATION
8:     Find the best refinement  $c_{best} \in \rho(c)$ 
9:     Assign  $c := c_{best}$ 
10:  until Necessity stopping criterion is satisfied
11:  Add  $c$  to  $\mathcal{H}$  to get new hypothesis  $\mathcal{H}' := \mathcal{H} \cup \{c\}$ 
12:  Remove positive examples covered by  $c$  from  $\mathcal{E}_{cur}$  to
13:  get new training set  $\mathcal{E}'_{cur} := \mathcal{E}_{cur} - cv(\mathcal{B} \cup \mathcal{H}', \mathcal{E}_{cur}^+)$ .
14:  Assign  $\mathcal{E}_{cur} := \mathcal{E}'_{cur}$  and  $\mathcal{H} := \mathcal{H}'$ .
15: until Sufficient stopping criterion is satisfied.
16: output: Hypothesis  $\mathcal{H}$ .
```

subsumption and Buntine (1988)'s *generalized subsumption*. Both reduce to θ -subsumption when the background theory is empty. Generalized subsumption can be computed by applying *saturation* and standard θ -subsumption. Saturation of a query is to extend the conjunction with the heads of the background rules whose body matches with the conjunction itself, i.e. a forward chaining completion of the query (see more on this in Chapter 6).

4.3.2.3 LEARNING AS SEARCH

The purpose of a learning task that involves symbolic representations is to find a hypothesis – expressible in the symbolic language – that satisfies a given *quality criterion*. Such a criterion is typically expressed in terms of *coverage* of the data supplied. The data will typically consist of logical facts or interpretations and coverage is often equivalent to logical entailment. A key point is to see ILP as a *search problem* in a hypothesis space spanned by logical structures (e.g. clauses), structured by a *generality order* (i.e. the θ -subsumption lattice). *Search operators* modify the logical structure of the current hypothesis, thereby moving through the generality order, guided by *declarative bias* and *search bias* mechanisms to make the search more efficient. We will now briefly consider each of these aspects.

θ -subsumption provides the basis for two important ILP techniques. One is *bottom-up* building of *least general generalizations* from training examples, and the other is *top-down* searching of *refinement graphs*. Searching happens in a space ordered by θ -subsumption. Most algorithms search for refinements of clauses, but these clauses can be embedded in a more biased form such as trees and decision lists (see Section 4.2.3.1). The mentioned (deterministic) search, whether top-down, bottom-up, or a combination thereof, may employ any standard AI search technique, such as depth-first or breadth-first, beam-search and heuristic-driven search. Besides this, the search may be randomized in various ways.

A generic ILP algorithm for top-down learning is depicted in Algorithm 7. Its general structure of separate-and-conquer learning is present in many ILP systems. The outer loop is a *set covering* approach. For reasons of tractability, most ILP systems do not refine entire theories, but single clauses, that are subsequently added to the partial theory. After adding

each clause refined in the same manner as described above, the subset of positive examples covered by the current partial theory is removed and another clause is constructed for the remainder of examples. Adding a clause to the current theory will not make any of already-covered positives underivable. The inner loop is a *general-to-specific* search for clauses. The construction of a hypothesis assumes a current training set \mathcal{E}_{cur} (that is initially the complete training set \mathcal{E}) and the current hypothesis \mathcal{H} (initially the empty set of clauses).

In the clause construction (*specialization*) loop, the *necessity stopping criterion* decides when to stop adding literals to a clause. In the hypothesis construction (*covering*) loop, the *sufficiency stopping criterion* decides when to stop adding clauses to a hypothesis. In *hill-climbing* search, the algorithm keeps one 'best' clause and replaces it with its 'best' refinement at each specialization step, whereas in a *beam search*, multiple 'best' clauses are kept. The stopping criteria in both repeat loops are domain-dependent. In case of perfect data, no negative examples should be covered, though when data is imperfect or noisy, one might be satisfied if as few as possible negative ones are covered or maybe when almost (though not all) positive examples are covered.

Search Operators. Search operators in ILP travel the lattice spanned by the θ -subsumption ordering. Top-down learners start from the most general clauses and repeatedly refine (specialize) them until they no longer cover negative examples (or until all positive examples are covered). In contrast, bottom-up learners start from the most specific clauses (allowed by the language bias) – typically the examples themselves – and then generalize a clause until it cannot be generalized further without covering negative examples (or when all positive examples are covered). Mixed forms occur, but are less frequent (but see the GOLEM system (Muggleton and Feng, 1992)). Most ILP systems employ a top-down search using a θ -subsumption-based *specialization operator* (or: *refinement operator*).

DEFINITION 4.3.5 ► A **specialization operator** (or, **refinement operator**) ρ maps a clause C to a set of clauses $\rho(C)$ of specializations of C : $\rho(C) = \{C' \mid C' \in L, C' \preceq_{\theta} C\}$

Specialization operators typically employ two basic syntactic operations on a clause: **i)** apply a substitution to the clause, and **ii)** add a literal to the body. In the θ -subsumption lattice, a *refinement graph* is defined as a directed, acyclic graph in which the nodes are clauses and the arcs correspond to the basic refinement operations. For example, Figure 4.10 shows part of a *refinement graph* starting from the clause $\text{onTop}(X, Y) \leftarrow$. In one of the branches, the clause is extended with $\text{above}(X, Y)$, and in subsequent refinements, $\text{above}(X, Y)$ might be refined to $\text{above}(X, a)$ by applying the substitution $\{Y/a\}$. Note that when a clause gets longer, increasingly many refinements can be generated, in particular when the clause contains many variables.

Several complications arise in case of such refinements. *Infinite chains* might arise if one stays in the same equivalence class, for example by extending the clause $\text{onTop}(X, Y) \leftarrow \text{above}(X, Y)$ with $\text{above}(Z_1, Z_2)$, with $\text{above}(Z_3, Z_4)$ and so on. Furthermore, it is also possible that a (proper) refinement does not change coverage of the clause. This is called the *determinacy problem*. This, however, can be (partially) solved by using a *look-ahead* in the search. Lastly, Figure 4.10 shows two paths in the graph that arrive at the same clause. This redundancy in the search process visits the same clause via different refinement sequences. *Optimal refinement operators* solve this problem by ensuring unique paths to refined clauses.

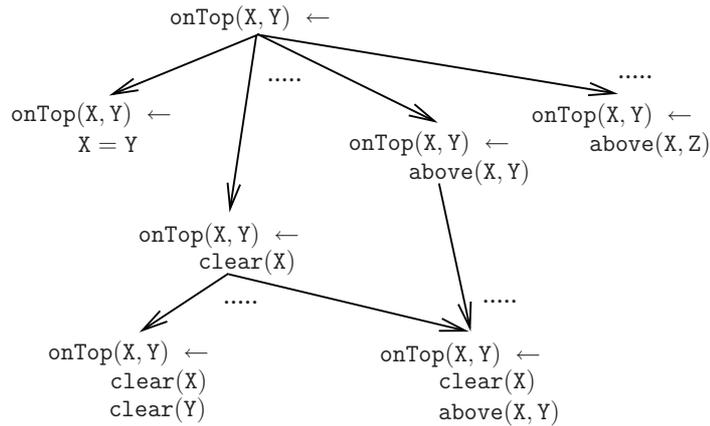


Figure 4.10: Part of the refinement graph for the relation $\text{onTop}(X, Y)/2$ (see Figure 4.6)

The opposite of specialization operators are *generalization* operators, that move in the opposite direction of the θ -subsumption lattice.

DEFINITION 4.3.6 ► A **generalization operator** ρ maps a clause C to a set of clauses $\rho(C)$ which are generalizations of C : $\rho(C) = \{C' \mid C' \in L, C \preceq_{\theta} C'\}$.

Generalization operators typically employ two basic syntactic operations on a clause: **i)** apply an inverse substitution to the clause, and **ii)** remove a literal from the body. In Figure 4.10, moving in the opposite direction of the arrows induces a generalization, e.g. $\text{onTop}(X, Y) \leftarrow \text{clear}(X)$ is a generalization of $\text{onTop}(X, Y) \leftarrow \text{clear}(X), \text{clear}(Y)$.

Computing the *greatest lower bound* of two clauses w.r.t. to the lattice is equivalent to computing the *least general generalization* (lgg), which is commonly used in ILP systems. These lgg's can be used for *bottom-up construction* of clauses, starting from the ground examples (see also Chapter 6).

DEFINITION 4.3.7 ► The **least general generalization** (lgg) of two clauses C_1 and C_2 , denoted $\text{lgg}(C_1, C_2)$, is the least upper bound of C_1 and C_2 in the subsumption lattice.

If a clause C θ -subsumes clauses C_1 and C_2 , it θ -subsumes $\text{lgg}(C_1, C_2)$ as well. The rules for computing the lgg of clauses make use of *inverse substitutions*. An inverse substitution θ^{-1} of a literal A is the replacement of terms of A with variables such that there is a substitution θ with $(A\theta^{-1})\theta \equiv A$. The lgg of two clauses is computed by a recursive anti-unification. Computing lgg's is an expensive process, exponential²² in the number of literals. Furthermore, because the resulting clause usually contains redundant literals, it must be reduced before further computations can happen. Again, as in the case of deciding subsumption, the use of an additional background theory in lgg-computations needs a more general setting, for example *generalized subsumption* (Buntine, 1988).

Declarative Bias, Heuristics and Efficiency. Because FOL is more expressive than propositional representations the search space explored by ILP systems is much larger (and often infinite). The combinatorics of the search space, the determinacy problem and the fact that the space is infinite in ILP systems, make it necessary to constrain the search space in some

²²For two clauses it amounts to the length of the first clause times the length of the second.

way. Useful constraints may also be induced by information about the current task and its representation. Consider a BLOCKS WORLD problem and a current hypothesis clause body $\text{on}(X, Y)$. Under θ -subsumption, it is possible to add $\text{on}(X, X)$ to the current hypothesis, although it is obvious that this does not make sense. The same holds for a refinement of $\text{on}(X, Y)$ with $\text{on}(\text{floor}, Y)$. Adding $\text{on}(W, W)$ should be avoided as well, for it creates two free variables, i.e. W is not connected to variables occurring in the current hypothesis. In general, there are many refinements that are known to be useless and they should be avoided. A variety of syntactic and semantic *declarative bias* mechanisms exist to restrict the number of rules considered during the search. Declarative bias and background theories both represent *partial knowledge* the user can have about a problem. Interestingly, background knowledge makes declarative bias even more necessary. A background theory extends the language with powerful abstractions, but at the same time, enlarges the search space (i.e. the number of possible refinements). Without this partial knowledge, learning is – in principle – still possible, though the search space to be explored will be much larger. The majority of systems employs a bias mechanism implemented by *mode* and *type declarations*. A type declaration restricts the types of queries and clauses to be type conform (i.e. the types of arguments of predicates). A mode declaration specifies properties of *calling patterns* in clauses, queries or conditions (e.g. a mode $\text{on}(+, -)$ states that when calling the predicate on the first argument should be bound whereas the second one should not be instantiated). Bias can also be used to search for *small* hypotheses, using *penalty terms*, heuristics on clauses lengths and various other means.

A third improvement of ILP systems resides in the use of *compact representations* and *efficient data structures and operations* employed in the induction process. Several approaches trade-off memory requirements and computation speed. A *query pack* is a set of similar queries structured into a tree; common parts of the queries are represented only once in such a structure. Query packs can be used to store a large number of candidate tests efficiently. The technique of *tabling*, implemented in some PROLOG systems, can be used to store results of SLDNF-derivations selectively, such that they can be reused (instead of recomputed) in subsequent queries (see also Chapter 6). Because Horn clause induction is a hard problem, it pays off to investigate the efficiency of the search process as a separate problem (see Struyf, 2004, for an extensive discussion on this topic).

ILP Systems as Lifted ML Systems. TILDE can be described as a first-order decision tree algorithm, but in a broader context, it is a *lifted* version of propositional algorithms (see Breiman *et al.*, 1984). In fact, a number of other propositional algorithms has been lifted to the first-order case. Many traditional ILP algorithms that aim at inducing logic programs (Lloyd, 1991) can be understood as lifted versions of propositional rule learners. Typical examples are PROGOL (Muggleton, 1995) and FOIL (Quinlan, 1990). Mooney and Califf (1995) learn first-order *decision lists* (see Section 4.2.3), i.e. ordered sets of rules.

A number of lifted algorithms has been introduced under the learning from interpretations setting, which is – as explained – the first-order setting that is closest to the traditional attribute-value (i.e. propositional) setting. van Laer and De Raedt (2001a) introduce a general methodology for upgrading propositional concept learning algorithms to the first-order case (see van Laer, 2002, for a thorough treatment). Some of the important steps in this methodology are: **i)** find the most suitable propositional algorithm, **ii)** use interpretations as examples, **iii)** use first-order literals as tests and adapt the coverage test, **iv)** use θ -subsumption as a framework for generality, and **v)** use an operator under θ -subsumption.

The first step in this methodology naturally provides a specific context in which all general aspects of ILP we have discussed, reside. As an example, take the tree learner TILDE (Blockeel, 1998). The tree setting entails that each hypothesis represents a first-order tree and that the refinement operators manipulate the leaf nodes of the current hypothesis tree. In TILDE, the *best refinement* in the *specialization* step in Algorithm 7 is a new internal node (i.e. a split). The best refinement corresponds to that test that splits the data covered by that part of the tree so far, most evenly (e.g. *information gain*) into two new leaf nodes (where the test fails or succeeds) where the goal is to make the sets of examples covered by each leaf node class-homogeneous (or as much as possible). The declarative bias is used to limit the number of candidate tests considered in each specialization step. Many heuristics and efficiency improvements can be implemented on top of the general algorithm, such as query packs and efficient multi-query subsumption procedures (e.g. see Blockeel *et al.*, 2000a; Blockeel, 1998; Struyf, 2004).

Among the algorithms that fit in van Laer (2002)'s methodology are TILDE, the concept (rule) learner ICL (see van Laer, 2002), the clausal theory learner CLAUDIEN (De Raedt and Dehaspe, 1997) and the clustering algorithms by Ramon (2002). Besides this specific setting, many other first-order algorithms for clustering, association rules and various forms of *data mining* have been proposed (see Džeroski and Lavrac, 2001b, for a recent overview). A large majority of ILP algorithms fits the general outline of the methodology, though they modify various aspects (such as the choice for a generality framework).

An additional class of methods consists of lifted versions of *genetic algorithms* (Goldberg, 1989). Because ILP is basically a search problem, evolutionary algorithms have been applied as an alternative for the standard top-down and bottom-up learning techniques. Divina (2004, 2006) surveys recent techniques and in Chapter 5 we will discuss this in more detail.

4.3.3 Statistical Relational Learning

The field of ILP has developed a variety of algorithms and theories on how to learn logical abstractions such as trees, rules, decision lists and logic programs. Most of these systems have been used in concept learning tasks where there is little (or no) noise in the data and where the goal of induction is some form of logic program. Many real-world problems however, contain noise, are probabilistic in nature, or even non-stationary. For these situations it becomes necessary to explicitly represent the *probabilistic* aspects of the task. Consider again a family relations example that is now augmented with blood type information. In this case, we would like to represent (and infer) that the probability of a person having blood type A depends probabilistically on its parent's blood types. Or in a citation database, it would be useful to learn, given certain characteristics of some paper and the authors, the probability that one paper will cite another. This can be based on *relations* between authors, papers, and other aspects such as keywords of a paper. Such tasks are highly probabilistic, often involve large datasets, and could not be handled by less rich representational formalisms such as propositional logic.

This recent probabilistic–logical direction is known under the general name *statistical relational learning* (Getoor and Taskar, 2007) and also under *probabilistic logic learning* (De Raedt and Kersting, 2003) and *probabilistic ILP* (De Raedt and Kersting, 2004). Its foundations are formed by probabilistic logics (see Section 4.2.2.3), but its strength lies in the combination of tractable logical representation and induction schemes (e.g. ILP) with

efficient, established learning algorithms in ML, especially graphical models. Its growth is witnessed by a number of recent events²³ and many applications of SRL can be found in *collective classification*, *link prediction*, *link-based clustering*, *social network modeling*, *bioinformatics*, *citation analysis* and many more. De Raedt (2008) provides an extensive – and very recent – treatment of ILP and SRL techniques developed so far in the literature.

SRL can be viewed upon from two angles. The first is to see SRL as an attempt to unify FOL with probability theory (PT), thereby extending the types of uncertainty from simple disjunction to full probability distributions. In the worst case, this is intractable (Halpern, 1990) though many systems have been developed that make inference and learning feasible for restricted first-order representations. A second view upon SRL is to see it as studying *lifted* versions of propositional, probabilistic representation formalisms and learning algorithms. For propositional representations, probabilistic models such as Bayesian networks and hidden Markov models are relatively well understood. Typical SRL systems focus on the supervised and unsupervised paradigms in ML. Relational RL can be seen as lifting the RL paradigm, and indeed it shares many problems and opportunities with SRL.

Many approaches in this new field of *probabilistic logic learning* (PLL) (De Raedt and Kersting, 2003, 2004) can be seen as upgrades (van Laer and De Raedt, 2001a) of propositional, probabilistic ML algorithms to the relational case, just like TILDE (Blockeel and De Raedt, 1998) can be seen as a logical upgrade of propositional tree-learning algorithms. The field of PLL has developed many methods that can deal with the intrinsic probabilistic nature of RL problems, such as relational upgrades of Bayesian networks (Kersting and De Raedt, 2001) and hidden Markov models (Kersting *et al.*, 2003), Markov Models (Anderson *et al.*, 2002), kernels (Cumby and Roth, 2003; Gärtner, 2003) naive Bayes classifiers (Flach and Lachiche, 1999), and probabilistic modeling methods such as *probabilistic relational models* (PRM) (Getoor *et al.*, 2001; Sanghai *et al.*, 2003), *probabilistic PROLOG* (De Raedt *et al.*, 2007b) and *Markov logic networks* (MLN) (Domingos and Richardson, 2004).

One intuitive SRL approach is *probabilistic ILP* (PILP), that frames the problem of learning first-order probabilistic concept representations in the ILP framework. The key point is to extend the notion of coverage to a probabilistic one:

DEFINITION 4.3.8 ► A **probabilistic covers relation** cv_{prb} takes as arguments an example e , a hypothesis \mathcal{H} and possibly a background theory \mathcal{B} and returns the likelihood of example e w.r.t. \mathcal{H} and \mathcal{B} . Formally, $cv_{prb}(\mathcal{H} \cup \mathcal{B}, e) = P(e \mid \mathcal{H}, \mathcal{B})$, where P denotes a probability distribution.

Probabilistic coverage can be implemented in each of the ILP settings defined in Section 4.3.2.1. Now a generic PILP problem can be stated as:

DEFINITION 4.3.9 ► **Probabilistic inductive logic programming** (De Raedt and Kersting, 2004) is defined as a probabilistic generalization of Definition 4.3.1.

Given:

- a set \mathbb{H} of possible programs,
- a set $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$ of positive and negative examples
- a consistent (definite) background theory \mathcal{B}
- a probabilistic covers relation $cv_{prb}(e \mid \mathcal{H}, \mathcal{B}, e)$

²³(Dietterich *et al.*, 2004), (Getoor and Jensen, 2004) and recent *Dagstuhl* seminars, e.g. see <http://www.dagstuhl.de/05051/index.en.shtml>

Find:

the hypothesis $\mathcal{H}^* = \arg \max_{\mathcal{H}} \text{cv}_{\text{prb}}(\mathcal{E} \mid \mathcal{H}, \mathcal{B})$

Each $\mathcal{H} \in \mathbb{H}$ consists of a *logical structure* \mathcal{L} (e.g. a definite program) and a set of (probabilistic) *parameters* ϱ . This naturally introduces two learning tasks: **i)** *parameter estimation*, where a fixed logical structure \mathcal{L} is assumed and the learning task is to estimate the corresponding parameters ϱ that maximize the likelihood (of the data), and **ii)** *structure learning*, where both the logical structure \mathcal{L} and the parameters ϱ have to be learned. Note that in the second task, one typically focuses on learning \mathcal{L} first, after which the parameters ϱ follow using the first task. This is in line with many types of tasks where structures and parameters have to be learned in parallel (see Chapter 3).

One recent approach that unifies many of the previous approaches are *Markov Logic Networks* (MLN) by (Richardson and Domingos, 2006). An MLN is a set of pairs (φ_i, w_i) , where each φ_i is a FOL formula and w_i is a real value. Together, all formulas φ_i form the knowledge base KB. Given a set of constants representing objects in the domain, it defines a probability distribution over possible worlds, where each world is a first-order structure. The distribution is in the form of a *log-linear model*: a normalized exponentiated weighted combination of features of the world. Each feature is a grounding of a formula in the KB, with a weight attached. In FOL, formulas are *hard* constraints: a world that violates even a single formula is impossible. In Markov logic, formulas are *soft* constraints: a world that violates a formula is less probable than one that satisfies it, other things being equal, but not impossible. The weight of a formula represents its strength as a constraint. Finite FOL is the limit of Markov logic when all weights tend to infinity. An existing first-order KB can be transformed into a probabilistic model simply by attaching weights to formulas. Weights can be set by design, or learned from additional data. Inference in MLNs is performed by MCMC²⁴ sampling over the minimal subset of the ground network required for answering the query. Additional clauses can be learned using ILP techniques (e.g. see Kok and Domingos, 2005).

MLNs offer a highly unconstrained representation; all that is needed is a knowledge base that contains formulas that are relevant to the current domain. In contrast, many other SRL methods impose a certain amount of structure on the logical representation. By doing this, specialized parameter learning methods become available. For example, *Bayesian Logic Programs* (BLP) (Kersting and De Raedt, 2001) model probability distributions over the set of ground atoms (which are viewed as random variables) using definite clauses with probabilities attached. In this way, BLPs have pure PROLOG (if no probabilities are specified) and Bayesian networks (if all rules are ground) as special cases, and in this way existing algorithms from both fields can be applied.

A related area is the use of relational upgrades of *neural networks* (e.g. Botta *et al.*, 1997; Kijssirikul *et al.*, 2001; Browne and Sun, 2001), and related combinations of *symbolic* and *sub-symbolic* methods in RL (Sun and Peterson, 1998; Sun, 1998). There are many challenges in representing knowledge for, and extracting knowledge from such first-order neural networks (Bader and Hitzler, 2005; Bader *et al.*, 2006). More specifically the mapping of first-order terms onto sub-symbolic nodes in the networks (other than by propositionalization), or the embedding of logic programs into such networks (Hitzler *et al.*, 2004), are important challenges. For relational RL these extensions could be very

²⁴Markov Chain Monte Carlo (e.g. see Hastie *et al.*, 2001).

useful, knowing that many successful applications in propositional RL are obtained using neural networks.

Although SRL approaches could prove very useful for relational RL algorithms, their use is still limited to only a couple of approaches. For example, Guestrin *et al.* (2003a) use PRMs to model structural, probabilistic dependencies and Croonenborghs *et al.* (2007a) use *probability trees* to capture *partial models*. The limited use of the achievements of SRL for relational RL is due to the fact that relational RL methods are still focused on Markovian environments (in contrast to POMDPs where the use of (relational) probabilistic models would be required) and that when probabilistic representations are needed – for example for the representation of transition models – most approaches use fairly simple probabilistic extensions of first-order action models such as STRIPS and situation calculus. In the near future it is expected that both fields will be more connected though, especially when relational POMDP representations and algorithms will be developed.

4.4. Acting in First-Order Domains

The notion of an *action* is a core component of various AI subfields such as intelligent agents, planning and commonsense reasoning. Theories of actions allow us to *specify* in an elaboration tolerant manner the relationship among objects, facts of the world that may change their values, or mental objects that encode the mental state of the agent and the effect of actions on all these matters and *reason* about them. This section is about formalizing actions and *change*. In Section 4.2 we have dealt with (first-order) logical representation, generalization and reasoning. Section 4.3 introduced inductive methods that learn logical abstractions from data. In the current section we will turn to the formalization of *actions* for dynamic first-order domains. These three parts together make up most of what we need to formalize *first-order Markov decision processes* in Section 4.5.1.

We will proceed as follows. First we describe the core components of formalizations of dynamic worlds: *fluents* and *actions*. Then we highlight some of the key challenges in modeling and axiomatizing dynamic worlds. We describe two example action formalisms – STRIPS and the situation calculus – in somewhat more detail to show some concrete implementations of important notions. Furthermore, these systems are important because they are used extensively in relational RL and much of it is reused in Chapters 5 and 6. Throughout the section we will be mainly concerned with deterministic planning domains, though at the end some useful extensions are briefly mentioned.

4.4.1 Formalizing and Modeling First-Order Domains

The last decades a great number and variety of FOL-based approaches have appeared for reasoning about action and change (see Gelfond and Lifschitz, 1998; Reiter, 2001; Russell and Norvig, 2003; Brachman and Levesque, 2004; Mueller, 2006). Despite many differences, most systems agree on the following aspects (Schwind, 1999). First, actions are usually specified by a pair of formulas $\langle \text{pre}, \text{post} \rangle$ where *pre* denotes the *preconditions* necessary for executing the action and *post* represents the effects of the action once it is executed. Second, actions occur in a state (or, *situation*) of the world. Consequently in action formalisms it is possible to talk about *actual*, *future* and eventually *past* world states by formulas. Third, most high-level action theories support the representation of *general laws* (or, *constraints*) that must hold in all situations and that can never change. Roughly,

we want to know what does and does not change in the world when we execute an action.

Research in reasoning about actions and change aims at providing frameworks for capturing dynamic domains. These are domains whose properties change as time passes by and events happen. Changes are usually assumed to be consequences of an event taking place. Events are divided into outcomes of the actions an agent deliberately performs, and *exogenous* events, that are natural phenomena beyond the control of the agent. This is the basic ontology for dynamical systems.

It is important to distinguish from the start between two types of action formalisms. The first type will be denoted *action languages* (see Gelfond and Lifschitz, 1998), which are KR schemes aimed at *representing* and *modeling* first-order domains and actions. Examples are STRIPS (Fikes and Nilsson, 1971) and ADL (Pednault, 1989) which allow for compact representations of problems with large sets of states and actions and for efficient *planning algorithms* (Weld, 1999) to compute plans in these domains. Action languages emphasize the aspect of *acting* and *planning* for concrete applications. The second type of action formalism will be denoted *action logics*. These formalisms are complete logical systems in which actions are *axiomatized* within the logic itself. Action effects, preconditions, ramifications and everything else is posed in a system of axioms, and one can infer all that matters as logical consequences of the logic itself (sometimes aided by second-order reasoning). Examples of action logics are the situation calculus (see Section 4.4.2), the *fluent calculus* (Thielscher, 1998) and the *event calculus* (Mueller, 2006). The main difference between these two types of formalisms is that action logics are more *declarative* and focused on *logical reasoning* whereas action languages more *procedural* and focused on *acting* and *planning*. This distinction, however, is not an all-or-nothing matter. Characterizations of the formal semantics of action languages such as STRIPS (e.g. see Lifschitz, 1986) and the use of action logics for planning purposes (e.g. see Reiter, 2001, Chap. 15)²⁵ are examples of combinations of approaches (see also Baral and Tran, 1998). Still, the distinction between these two types of formalism is useful for understanding the difference between *representing* actions and *reasoning about* actions.

4.4.1.1 ACTIONS AND FLUENTS

Dynamic domains introduce two main types of *change*. One is *belief revision* (Gärdenfors, 1988); one might feel the need to rethink what one knows about the world. The second type concerns beliefs about a world that changes itself. The first type of change is an important topic, though we will ignore it in this book (though there are relations with POMDPs and belief states). We will mainly be occupied with the second type. We need formalisms for modeling dynamic worlds in which actions can be taken that modify parts of these worlds, e.g. for transition functions over first-order domains.

Actions are first-order terms, consisting of an action *name* and its *arguments*. For example, the action `move(a, b)` has two arguments and has the intuitive meaning of moving block `a` on block `b`. Actions change the world as it is now, and more specifically, they change *fluents*. *Fluents* are first-order terms, denoting properties of the domain that can change their value over time. *Relational* fluents are ground, relational atoms and *functional* fluents are ground, functor terms. For example if in the current world `on(a, c)` and `clear(b)` and `clear(a)` hold, the action `move(a, b)` will have an influence on the fluents

²⁵The title of this book *Knowledge in Action* is an interesting reference to *Knowledge in Flux* by Gärdenfors (1988). Both books are about the dynamics of mental states, though in very different ways.

on(a, c), clear(b) (i.e. their truth value will change), but not on clear(a).

Most action formalisms assume that the world satisfies at least the following basic criteria: **i)** only one action can occur at a time, **ii)** actions are atomic, i.e. effectively instantaneous, and **iii)** nothing changes except as the result of planned actions, either directly or indirectly. For example, the action move(a, b) in our example will directly influence some of the fluents, but will indirectly influence fluents such as inSameStack(a, b) which was false before executing the action, but true afterwards.

Characterizing a dynamic first-order domain is all about specifying, representing or axiomatizing how actions influence the values of fluents, how fluents influence each other, and which fluents are subject to change and which are not. A *causal theory* is comprised of several classes of rules or axioms that characterize exactly this. For a general understanding one can assume that these causal theories are defined on transitions between first-order structures, i.e. the *states* of the world. In the following, we will describe the main patterns of fluent dynamics and how to formalize them.

4.4.1.2 FUNDAMENTAL CHALLENGES

Independently of the way the dynamics of a particular domain are modeled or axiomatized, a number of fundamental problems arises that any sufficiently powerful action formalism has to address in some way. Most of these problems have to do with what changes and what does not, what exactly influences these changes, and how changes accumulate over time. Merely representing these notions is a non-trivial task, and logically axiomatizing them has been the topic of much research and it has generated many logical systems and solutions. Note that the descriptions we give here are meant to convey the conceptual ideas. The technical implementation in a logical system is often found to be challenging and various solutions can be found in current action logics. We will see some more concrete examples in STRIPS and the situation calculus later in this section.

Causality and Constraints. The fundamental property of dynamic worlds is that aspects of this world change over time, but that some aspects will always hold. Quite naturally the question arises about what *causes* these changes. A long time ago, Leibniz' addressed this question with his *principle of sufficient reason*: "Nothing at all happens without some reason". A very general taxonomy of causes used in many action formalisms includes at least **i)** changes caused *explicitly* by actions, **ii)** changes caused implicitly by actions, due to *ramifications*, *domain constraints* or *general laws*, and **iii)** (non-deterministic) changes caused by external factors. The third type of cause are called *exogenous events* and these cover all causes other than the first two. We will only deal with the first two types.

Actions are often the main cause of changes in the world. A general formulation of such an *effect axiom* is the following:

$$a \text{ causes } f \text{ if } f_1, \dots, f_n \quad (4.6)$$

where the action a will have an effect on a fluent f only if fluents f_1, \dots, f_n hold in the context where the action is executed. Fluents f_1, \dots, f_n determine the preconditions of a and are part of the *qualification problem* (see further in this section). Furthermore, they can provide a template context for the action *parameters* and its effects for the execution of a concrete action. For example, an action move(X, Y) can only be executed when the variables X and Y are instantiated, and for a possible effect (i.e. f in the rule) on(X, Y) this holds too.

The second case of *indirect* effects of actions is usually more generally formalized as a causal relationship between fluents themselves, such as in f_1, \dots, f_n **causes** f (**if** g_1, \dots, g_k) where $g_1 \dots g_k$ provides an optional context. For example, when a bag is moved from room 1 to room 2 by executing an action, every item that was in the bag is now in room 2 too. Typically, the action specification only mentions that it will change the location of the bag, and a possible *ramification* is given by the rule $\text{in}(\text{room2}, \text{bag})$ **causes** $\text{in}(\text{room2}, \text{item})$ **if** $\text{inside}(\text{item}, \text{bag})$. Other indirect effects are triggered by domain rules (or, *common laws of inertia* (Mueller, 2006)) which state relationships between fluents that always hold.

Note that we have encountered similar patterns when describing *factored dynamics* in Section 3.5. Edges between time slices in a DBN correspond to direct effects, whereas dependencies within a slice (i.e. *asynchronic arcs*) correspond to indirect effects. And, similar to the DBN setting, indirect effects require additional computational efforts; once the direct effects have been computed, causal laws such as the ones in the above are used to derive additional effects. When causality is represented explicitly, various forms can be distinguished (Schwind, 1999). For example, it can be represented as an inference relation on normal FOL formulas which can be characterized by axioms and inference rules. Another form is to use a non-logical predicate $\text{causes}(f, v, s)$ meaning that the fluent f is caused to get the value v in state s . Here we will have special non-logical axioms for specifying properties of this predicate.

State constraints, or *integrity constraints* are general laws that hold in the domain. If we take φ and ψ to be formulas then two general forms are $\varphi \rightarrow \psi$ and $\varphi \leftrightarrow \psi$. These laws are static and must hold always. We have informally encountered such laws when introducing RMDPs in Section 4.1.3.2 where we used them to discard illegal states. An example of such a law is $\forall X \neg \text{on}(X, X)$, which says that no block can be on itself. State constraints are important for axiomatizing a domain, and in Chapter 6 we make use of it for model-based relational RL.

The Frame Problem. The core problem of dealing with actions and causality is the *frame problem*²⁶: determining which relations remain *unchanged* after the execution of a single

²⁶One of the nicest explanations of the frame problem was given by Dennett (1998). Although it does not explain the technical difficulties that arise when approaching it in a concrete action logic, it does give a good account of the conceptual idea. A robot R1 is introduced that has to save a vital battery that is locked inside a room with a bomb. The robot computes a plan to get a key, go to the room and rescue the battery. The battery is located on a wagon but the problem is that R1 pulls the wagon out, which holds the bomb as well. R1 knew that the bomb was on the wagon too, but failed to recognize the fact that by pulling the wagon out, carrying the battery, the bomb would still be on there. And unfortunately, the bomb explodes. A second robot is then developed, called R1D1 and this one is endowed with the capability of inferring all implications of its actions, and implications of these implications and so on. So, when computing a plan to rescue the battery, it also computed whether the removal of the wagon would entail also moving the bomb with that. But, unfortunately, it also inferred whether the color of the room would change because of its actions, and whether the wheels would turn more revolutions than there are wheels on the wagon and so on. R1D1 would never complete the final plan, because it kept inferring new facts indefinitely... and the bomb went off. A third robot was invented, called R2D1 that would learn from this mistake, and this one would only infer *relevant* facts. Again, there was a problem. R2D1 was so busy inferring new facts and deciding upon their relevance that again, it was thinking indefinitely... and the bomb went off. This characterizes the frame problem quite nicely: deciding upon (and representing) which aspects of the world change and which parts remain unchanged when executing actions, is in general undecidable. There are many, technical solutions to make the problem more tractable, but it remains a deep problem in AI and philosophy. Dennett cites a researcher saying "We have given up the goal of designing an intelligent robot, and turned to the task of designing a gun that will destroy any intelligent robot that anyone else designs."

action. For example, the movement of a block will change its relations with other blocks, but presumably not its color, shape or weight. A general assumption is that actions will change only a relatively small number of relations, but given the vast amount of possible relations, the challenge is to represent the invariants compactly and efficiently. An example *frame axiom* would specify that when a block's color is blue, it will still be blue after a move action. If there are $\#A$ actions and $\#F$ fluents, we typically have about $2 \cdot \#A \cdot \#F$ frame axioms. Reiter (2001, p. 23): "A solution to the frame problem is a systematic procedure for generating, from the[se] effect axioms, all the frame axioms. If possible, we also want a parsimonious representation for these frame axioms." Several action formalisms support sophisticated axiomatizations of the frame problem (e.g. the situation calculus) whereas others use simpler mechanisms in spirit of CWA; everything not explicitly mentioned, remains unchanged²⁷ (e.g. STRIPS). Examples are provided later in this section.

Representing action effects in a compact way and *inferring* truth values of fluents when executing (several) actions are both aspects of the general frame problem. The first aspect is known as the *representational frame problem* (RFP). Let $\#F$ be the number of fluents, $\#A$ be the number of actions and let $\#E$ be an upper bound on the number of action effects for each action. A *frame axiom* specifies explicitly what *does* stay the same when executing an action. This results in $O(\#A \cdot \#F)$ frame axioms. However, it is assumed that this can be done more compactly since each action only affects *some* fluents, i.e. because usually $\#E \ll \#F$. Going from $O(\#A \cdot \#F)$ to $O(\#A \cdot \#E)$ is the RFP.

The prediction problem is the problem of predicting which relations hold *after a sequence of actions*, e.g. a *plan*. Doing this efficiently is called the *inferential frame problem* (IFP). Consider again $\#A$, $\#E$ and $\#F$ and let t be the number of steps (i.e. actions). Projecting the result of a t -step sequence of actions involves looking at each of the $\#F$ frame axioms with average size $\frac{\#A \cdot \#E}{\#F}$ which amounts to $O(\#A \cdot \#E \cdot t)$ inferential work. However, most of this work is comprised of copying the values of unchanged fluents. Moving from $O(\#A \cdot \#E \cdot t)$ (or from $O(\#F)$) to $O(\#E \cdot t)$ is the IFP. *The persistence problem* is complementary to the prediction problem and is about predicting which relations will remain *unchanged* after a sequence of actions.

The Ramification Problem. The problem of representing and reasoning about the indirect effects of events is known as the *ramification problem*. For example, the relation $\text{onTop}/2$ defined in Figure 4.6 holds for blocks B_1 and B_2 if in the current state B_1 is the top block of a stack which holds B_2 too. However, configurations of stacks vary from state to state and changes are caused by actions that affect the $\text{on}/2$ relations explicitly. The indirect effect of moving b_1 to the top block of a stack which contains B_2 has the indirect effect of making $\text{onTop}(B_1, B_2)$ true. In planning systems, a relation $\text{on}/2$ is usually called a *primitive predicate* and the $\text{onTop}/2$ a *derived predicate*. More specifically, Reiter (2001, p.402): "solving the frame problem when given a set of effect axioms and state constraints as initial data is called the ramification problem". Solutions are numerous and vary from formalism to formalism (see Mueller, 2006, Chap. 6 for a good overview).

Similar techniques are found in *database logics*, which use the same mechanisms for updating states (i.e. databases) based on actions (i.e. update actions), where these actions can trigger further updates based on ramifications (i.e. derived updates) and where states and actions can be subject to constraints (i.e. integrity constraints). The connection be-

²⁷Pfeifer and Scheier (1999, p. 67) call this the *sleeping dog strategy*.

tween updates, constraints and derived predicates is made clear by Spruit (1994): "When an update is performed the derivation rules may lead to derived updates but cannot make updates fail. By contrast, constraints never lead to derived updates but can make updates fail". We will see examples of this in Chapter 6.

Many systems, however, do not support separately defined indirect effects, but instead *compile* these effects into the action descriptions, making all effects direct, i.e. eliminating all derived predicates by making them primitive. This is only feasible in case the number of indirect effects is small, because this compilation process generates large action descriptions. Schwind (1999), Reiter (2001, Appendix B) and Mueller (2006, Chap. 6) provide good descriptions of many technical solutions. Lang *et al.* (2003) focus on the computational complexity and show that even for the simplest action formalisms dealing with causality becomes intractable.

The Qualification Problem. Effect axioms describe which effects actions have, but equally important are the conditions under which an action can be applied. These conditions (the f_1, \dots, f_n in Equation 4.6) are called the *preconditions* of the action. Many systems have a separate *precondition axiom* for each action. Now the *qualification problem* is about the (im)possibility of listing all necessary preconditions required for a (real-world) action to have its intended effects²⁸. For example, moving block X onto block Y will succeed, *unless* block X is glued onto some other block Z, it explodes when touched, is too heavy to lift, etc. Listing all preconditions for the action $\text{move}(X, Y)$ as e.g. $\neg \text{willExplode}(X)$ is generally infeasible. There are obvious relations with *default logic*; e.g. when we do not know that blocks can explode, we will assume that they do not. Usually a trick similar to the generation of frame axioms is used to derive a parsimonious representation of all necessary preconditions for each action, in the presence of state constraints. However, similar to the ramification problem, the state constraints introduce additional problems when trying to axiomatize all these preconditions (see Reiter, 2001, Appendix B).

4.4.2 Two Characteristic Systems

Confronted with an enormous variety of logical action languages (Russell and Norvig, 2003; Brachman and Levesque, 2004; Mueller, 2006) we can only scratch the surface of this topic. Here we highlight some of the characteristics of two action languages that can be positioned at two distinct ends of a wide spectrum of languages. One end is formed by the STRIPS language, which is a simple action language aimed at *modeling* planning systems and action effects using a language with limited expressiveness but excellent computational properties. The other end is formed by the *situation calculus*, which is an expressive formalism for *axiomatizing* first-order action domains. Both tackle – to some extent – the problems addressed in the previous section, though in very distinctive ways. The distinction between modeling and axiomatizing (see McAllester, 1999) has implications for how much general reasoning and theory development is possible using the action theory as a logical system.

Both languages are central in the field of relational RL, and together they fit the pattern we have encountered in the sections on FOL and inductive logic. That is, in Section 4.2 we have encountered FOL as a general language, and we discussed Horn logic as a useful, efficient subset. In Section 4.3 we have discussed the general problem of learning logical

²⁸See also the informal introduction by Dennett (1998, p. 197–198).

structures but placed an emphasis on the use of limited subsets of FOL (e.g. Horn logic and ILP methods) for efficient algorithms. Similarly, the situation calculus can, again, be seen as the full, expressive formalism, whereas STRIPS represents a limited subset displaying better computational properties. At the end of this section we will discuss similarities and differences in a little more detail.

4.4.2.1 STRIPS

The STRIPS²⁹ representation (Fikes and Nilsson, 1971) is one of the simplest action formalisms we will encounter in this book. It was developed in the context of planning systems, though it appears in more general contexts and many relational RL systems make use of it. Furthermore, it forms a basis for a number of other, more expressive, action formalisms (see Russell and Norvig, 2003, and the end of this section). STRIPS is expressive enough to describe a wide variety of problems, but restrictive enough to allow for efficient computations in planning problems. In STRIPS, actions are not represented explicitly as part of a model of the world. As a consequence it is not possible to *reason* about them directly. Instead, actions are to be viewed upon as *operators* that modify the world models in a syntactic way.

In first-order³⁰ STRIPS the world is essentially modeled as an interpretation, for a given language. Since there are only positive literals (i.e. ground atoms) in the states, the closed world assumption applies. Actions are specified by extra-logical constructs:

DEFINITION 4.4.1 ► A STRIPS **operator** is a tuple $\langle \text{pre}, \text{action}, \text{add}, \text{del} \rangle$, where *pre* is a set of atoms describing the preconditions that must hold for the action to apply, and *action* is an action atom. The post-conditions are described by the add list *add*, which is the set of atoms that is added to the state and the delete list *del* which is the set of atoms that is deleted from the state.

Note that the effects (i.e. the *add* and *del* sets) of an operator must be read as conjunctions. Furthermore, they are sometimes represented as one general goal, e.g. $p \wedge \neg q$ meaning that *p* should be added and *q* deleted. An operator is a *template* (or, *schema*) over object variables, possibly resulting in multiple, ground instantiations of actions (and their effects). STRIPS assumes that the number of outcomes is relatively small and that there are no complex action effects, such as involving quantification. Furthermore, all aspects of an operator are implicitly existentially quantified (e.g. similar to clauses, see Section 4.2.2.1) and functor-free.

DEFINITION 4.4.2 ► The **operational semantics** of a STRIPS operator $O = \langle \text{pre}, \text{action}, \text{add}, \text{del} \rangle$ is defined as follows. Let *s* be a world model, i.e. a set of ground atoms. Applying *O* to *s* first implies finding a substitution θ (e.g. by resolution) such that $\text{pre}\theta \subseteq s$. Then the resulting state is $s' = (s \setminus \text{del}\theta) \cup \text{add}\theta$.

A common assumption is that the *add* and *del* lists are disjoint such that the set union and difference operators can be used in any order. If an atom occurring in the add list is already present in the state, it is not added twice. Similarly, if an atom occurs in the delete list but not in the state, it is ignored. STRIPS operators represent a very simple solution to

²⁹STRIPS stands for STanford Research Institute Problem Solver.

³⁰The original formulation uses a propositional logic.

the frame problem: anything not explicitly mentioned in the operators description remains unaffected. In fact, this was the first explicit solution to the frame problem.

EXAMPLE 4.4.1 ▶ An example STRIPS move operator for BLOCKS WORLD problems is the following:

action	move(X, Y)
precondition	cl(X), cl(Y), on(X, Z)
delete list	cl(Y), on(X, Z)
add list	cl(Z), on(X, Y)

Let us consider the application of this action to the state

$$s = \{\text{on}(a, b), \text{cl}(a), \text{on}(b, c), \text{on}(c, \text{floor}), \text{on}(d, \text{floor}), \text{cl}(d)\}$$

One possible variable substitution is $\theta = \{X/a, Y/d, Z/b\}$ resulting in

action	move(a, d)
precondition	cl(a), cl(d), on(a, b)
delete list	cl(d), on(a, b)
add list	cl(b), on(a, d)

such that the precondition is fulfilled, i.e. all the atoms occurring in the precondition hold in state s . The resulting state is computed by adding all the atoms in the add list and deleting all the atoms in the delete list from state s , resulting in the new state

$$s' = \{\text{cl}(a), \text{on}(b, c), \text{on}(c, \text{floor}), \text{on}(d, \text{floor}), \text{cl}(b), \text{on}(a, d)\}$$

A STRIPS *planning* problem is characterized by $\langle s_0, \text{OPS}, \text{GOAL} \rangle$, where s_0 is a list of ground atoms called the *initial state*, OPS a set of operators (see Definition 4.4.1) and GOAL a ground goal query. A solution (i.e. a *plan*) is a sequence $\langle O_1, \theta_1, \dots, O_n, \theta_n \rangle$ where each $O_i \in \text{OPS}$ and each θ_i is the result of applying O_i to the previous state (starting in s_0 , following Definition 4.4.2). This results in a sequence of states $\langle s_0, \dots, s_n \rangle$ with $\text{GOAL}\theta \subseteq s_n$ for some θ . This process is called *progression*; it progresses from the initial state to the goal, by repeatedly applying the operators. Another view is to see it as a deduction process, as a "proof" of the goal starting from the initial state using the operator definitions as a kind of axioms.

Since the introduction of STRIPS, characterizing its formal, *logical* semantics has been problematic. Several proposals have appeared in the literature (e.g. see Lifschitz, 1986) (see also the bibliographic remarks in (Reiter, 2001, Chap. 9)) and extensions have appeared that try to axiomatize STRIPS systems. The reasons for these difficulties is that although these systems appeal to concepts from logic (e.g. entailment, logical formulas), they are not axiomatizations in any logic. For example, actions and effects are not axiomatized and the operator descriptions appeal to extra-logical ideas. Nevertheless, for the purposes of relational RL, and with that, this book, STRIPS has been shown to be sufficient for modeling and representing actions effects. In Reiter (2001, p. 232)'s words; "... a STRIPS operator is a mapping from first-order structures to first-order structures, where the mapping is defined by the addition and deletion of tuples applied to the relations in the structures". An intuitive semantics can be defined in terms of *transition systems*³¹ (see Gelfond

³¹Note the similarities with possible worlds semantics and the accessibility relation present in modal logics, see Section 4.2.2.3.

and Lifschitz, 1998), i.e. by a directed graph where the nodes denote relational structures and an edge between nodes s_i and s_j represents the existence of an operator that maps s_i into s_j . However, there is no inherent restriction built in STRIPS that prevents it from entering illegal configurations; this must be enforced by ensuring that the operator definitions do not generate illegal states. Furthermore, while the sets of states and actions can be quite large, they are all instantiated from very compact operator definitions which hold for any domain instantiation.

What concerns the fundamental problems described in the previous section, one can say that STRIPS systems solve the prediction, persistence and (inferential and representational) frame problems by the operational semantics defined in Definition 4.4.2. The qualification problem has an equally simple solution: everything that should hold for an action to be applied is represented by the preconditions. Ramifications however, pose a problem. Implicit effects of actions cannot easily be represented. An often used solution is to *compile* all the (i.e. including implicit) effects into the operator, thereby enlarging these descriptions considerably.

STRIPS forms the base representation for many action modeling and planning systems. Pednault (1989) introduced the language ADL which relaxes some of the restrictions of STRIPS. ADL supports, among other things, both positive and negative literals in states, conjunctions, disjunctions and quantified variables in goals and furthermore conditional effects. Another variation on STRIPS is *functional STRIPS* (Geffner, 2000; Wang and Schmolze, 2005) that limits the language to only include functional fluents. Systems such as STRIPS and ADL have recently been standardized resulting in the *problem domain description language* (PDDL) (Ghallab *et al.*, 1998). Several subsequent versions of PDDL exist, adding support for e.g. *numeric functional fluents*, *time* and *duration*. Many other *action languages* exist, (see Gelfond and Lifschitz, 1998; Mueller, 2006; Brachman and Levesque, 2004, for overviews).

4.4.2.2 SITUATION CALCULUS

The *Situation Calculus* (McCarthy, 1963; Reiter, 2001) (SC) is a first-order language for axiomatizing dynamic worlds. It was the earliest treatment of time and action in AI. SC differs greatly from STRIPS in that it provides a fully first-order language that supports reasoning over *all* domain instantiations, even when the domain instantiation is unknown. SC consists of three *sorts*, which are *actions*, *situations* and *fluents*.

Actions are first-order terms consisting of an action function symbol and its arguments, similar to STRIPS. A *situation* is a first-order term denoting a *sequence of actions*. These are represented using the binary function symbol *do*, such that $do(\alpha, s)$ denotes the sequence resulting from adding the action α to the sequence s . A special constant S_0 denotes the *initial situation*, i.e. the empty action sequence. For example, the situation term

$$do(\text{move}(X, Y), do(\text{move}(Z, W), do(\text{move}(A, B), S_0)))$$

denotes the situation after applying three *move* actions, starting in the initial situation. Relations whose truth values vary from state to state are called *relational fluents* and are denoted by predicate symbols whose last argument is a situation term. For example, $on(X, Y, s)$ is the relational fluent meaning that block x is on block y after performing the action sequence s . *Functional fluents* can be defined analogously³².

³²Note that functional fluents can be specified as relational fluents. For example, the functional fluent

A domain theory is axiomatized in SC with four classes of axioms. First, *action precondition axioms* specify which conditions must hold prior to the execution of an action. There is one axiom for each action function $A(\vec{X})$, with syntactic form

$$\text{Poss}(A(\vec{X}, s)) \equiv \Pi_A(\vec{X}, s)$$

Here, $\Pi_A(\vec{X}, s)$ is a formula with free variables among \vec{X}, s that characterizes the preconditions of action A . Second, *successor state axioms* (SSA) describe how fluents change as an effect of actions. There is one such axiom for each fluent $F(\vec{X}, s)$ with syntactic form

$$F(\vec{X}, \text{do}(a, s)) \equiv \Phi_F(\vec{X}, a, s)$$

where $\Phi_F(\vec{X}, a, s)$ is a formula with free variables among a, s, \vec{X} . These characterize the truth values of the fluent F in the next situation $\text{do}(a, s)$ in terms of the current situation s , and they embody a solution to the frame problem (see Reiter, 2001). SSAs are constructed by first specifying *positive and negative effect axioms* that describe how actions influence the truth³³ value of a fluent, i.e. when it becomes true or false:

$$\begin{aligned} \gamma_F^+(\vec{X}, a, s) &\rightarrow F(\vec{X}, \text{do}(a, s)) \\ &\text{and} \\ \gamma_F^-(\vec{X}, a, s) &\rightarrow \neg F(\vec{X}, \text{do}(a, s)) \end{aligned}$$

Combining both positive and negative effects, one obtains a closed form of all the influences on some fluent F as an SSA:

$$F(\vec{X}, \text{do}(a, s)) \equiv \Phi_F(\vec{X}, a, s) \equiv \gamma_F^+(\vec{X}, a, s) \vee \left(F(\vec{X}, s) \wedge \neg \gamma_F^-(\vec{X}, a, s) \right)$$

In other words, the truth value of a fluent F is true after applying some action a either when a makes F true, or when F was already true and a did not make it false. Similar to STRIPS formalizations, the assumption is that only few actions will affect the truth value of a fluent. Because the length of a SSA is roughly proportional to the number of actions that affect the fluent, it is likely that the SSA remains short. Third, *unique names axioms for actions* state that the actions of the domain are pairwise unequal. Last, the *initial database* is a set of first-order sentences whose only situation term is S_0 and it specifies the *initial state* of the domain.

EXAMPLE 4.4.2 ▶ Let a domain axiomatization contain the actions $\text{move}(X, Y)$ and $\text{moveToFloor}(X)$ with their intuitive meaning. Let $\text{clear}(X, s)$, $\text{on}(X, Y, s)$, $\text{onFloor}(X, s)$ be relational fluents. The action precondition axioms are given by

$$\begin{aligned} \text{Poss}(\text{move}(X, Y), s) &\equiv \text{clear}(X, s) \wedge \text{clear}(Y, s) \wedge X \neq Y \\ \text{Poss}(\text{moveToFloor}(X), s) &\equiv \text{clear}(X, s) \wedge \neg \text{onFloor}(X, s) \end{aligned}$$

The *successor state axioms* are defined for all relational fluents as

$f(X, s) = v$ can be modeled as $f(X, v, s)$ with an additional axiom specifying that this mapping is a proper function definition.

³³Functional fluents are handled analogously.

$$\begin{aligned}
\text{clear}(X, \text{do}(a, s)) &\equiv (\exists Y)\{[(\exists Z)a = \text{move}(Y, Z) \\
&\quad \vee a = \text{moveToFloor}(Y)] \wedge \text{on}(Y, X, s)\} \\
&\quad \vee \text{clear}(X, s) \wedge \neg(\exists Y)a = \text{move}(Y, X) \\
\text{on}(X, Y, \text{do}(a, s)) &\equiv a = \text{move}(X, Y) \vee \\
&\quad \text{on}(X, Y, s) \wedge a \neq \text{moveToFloor}(X) \wedge \neg(\exists Z)a = \text{move}(X, Z) \\
\text{onFloor}(X, \text{do}(a, s)) &\equiv a = \text{moveToFloor}(X) \vee \\
&\quad \text{onFloor}(X, s) \wedge \neg(\exists Y)a = \text{move}(X, Y)
\end{aligned}$$

Furthermore, one specifies uniqueness of actions, using *unique name axioms for actions*

$$\begin{aligned}
\text{move}(X, Y) &\neq \text{moveToFloor}(Z) \\
\text{move}(X, Y) = \text{move}(X', Y') &\rightarrow X = X' \wedge Y = Y' \\
\text{moveToFloor}(X) = \text{moveToFloor}(X') &\rightarrow X = X'
\end{aligned}$$

Reasoning in SC is mainly done through *regression*. The regression of a formula φ through an action a is a formula φ' that holds prior to a being performed iff φ holds after a . SSAs support regression in a natural way. Suppose the fluent F 's successor state axiom is $F(\vec{X}, \text{do}(a, s)) \equiv \Phi_F(\vec{X}, a, s)$. We inductively define the regression of a formula whose situation arguments all have the form $\text{do}(a, s)$ as follows³⁴:

$$\begin{aligned}
\text{REG}(F(\vec{X}, \text{do}(a, s))) &= \Phi_F(\vec{X}, a, s) & \text{REG}(\neg\varphi) &= \neg\text{REG}(\varphi) \\
\text{REG}(\varphi_1 \wedge \varphi_2) &= \text{REG}(\varphi_1) \wedge \text{REG}(\varphi_2) & \text{REG}((\exists X)\varphi) &= (\exists X)\text{REG}(\varphi)
\end{aligned}$$

The idea is that regressing a query $\varphi[\text{do}(A_1, \dots, A_n), S_0]$ (i.e. asking whether φ holds in the situation after applying the sequence of actions $[\text{do}(A_1, \dots, A_n)]$ in the initial state) yields a formula that can be evaluated, using first-order theorem proving, with reference only to the initial database. Regression is an old technique in AI (see Waldinger, 1977) and in Chapter 6 we will deal extensively with this topic. In SC it is a convenient way to reason over sequences of actions, i.e. situations, and it provides a solution to the representational frame problem. However, the complexity of regression is at least linear in the number of actions (and sometimes worse), and this can become expensive for long execution traces. Moreover, this is being done again and again when reasoning (see further Thielscher, 1999). The inferential frame problem is not solved by this (but see Russell and Norvig, 2003, Chap. 10 for solutions).

The SC has been extended in several directions, addressing combinations with probability (Grosskreutz and Lakemeyer, 2000) and utility (Bacchus *et al.*, 1999; Boutilier *et al.*, 2000b), high-level constructions in the programming language GOLOG (Levesque *et al.*, 1997), *non-Markovian control* (Gabaldon, 2000), and *online* versions (Ferrein *et al.*, 2004; Soutchanski, 2001). Furthermore, *multi-agent*, *game-theoretic* extensions (Finzi and Lukasiewicz, 2004a) and various *epistemic* aspects (see modal logic, Section 4.2.2.3) have been studied in the SC (see also Reiter, 2001). The use of the SC for solving first-order, decision-theoretic problems (i.e. relational RL) will be discussed later in this book.

4.4.2.3 DISCUSSION

Both SC and STRIPS are central action representation formalisms in AI. We have seen that, although on the surface there are many similarities, they are quite different. The expressivity of STRIPS is very limited compared to SC. The basic modus operandus in SC

³⁴We show only a sufficient set of connectives. The rest is defined analogously.

is regression whereas for STRIPS it is progression. However, both systems can be used in various other ways. Furthermore, domain specifications can often be transformed from one formalism to another (see Reiter, 2001; Lin, 2003)³⁵. SC is a general logical language, amenable to various sorts of logical reasoning because all components (i.e. actions, relations, action preconditions) are defined in the logic itself. STRIPS systems can be characterized as *modeling* approaches whereas SC formalizations as *axiomatizations* (e.g. see McAllester, 2000). The latter are more powerful and support reasoning about actions in a FOL setting³⁶. The semi-logical notions of STRIPS do not support reasoning about the system itself (e.g. the actions) in a logical way, but instead provide a useful way for planning. Expressivity comes with a price (see Section 4.2.2), and it is a relevant question how much is needed for various tasks. Many planning systems operate mainly on propositional languages, or use relatively simple action languages such as STRIPS, ADL or PDDL (Russell and Norvig, 2003; Brachman and Levesque, 2004), because they are expressive enough for most specific tasks, i.e. realistic planning problems. For these problems, one typically focuses on computing solutions instead of logically reasoning about the system itself. Languages such as the SC are situated in the broader scope of *cognitive robotics*, which is the study of building control systems for robots (physical and virtual) using high-level commonsense knowledge representation and logical languages. Other expressive logics that share many similarities with the SC are the *fluent calculus* (FC) (Thielscher, 1998) and the *event calculus* (EC) (see Mueller, 2006). Compared to the SC, the FC introduces the notion of *states* and due to that it provides a solution to the inferential frame problem of the SC (see further Thielscher, 1999). The SC and FC are based on branching time, whereas the EC uses a linear time ordering but does not support functional fluents (see Mueller, 2006, p. 289 for some additional connections). Both the SC and FC have been used for model-based, relational RL (see Chapter 6).

Both SC and STRIPS formalizations, and indeed many action formalisms in general, have difficulties with *state constraints* and ramifications. Consider an action $\text{paint}(X, C)$ that paints an object X in color C . Its effects will specify that the relational fluent $\text{color}(X)$ will change. However, if the object X contains a *sub-part* Y , then Y will probably change color too. This is called an *implicit effect* of the action paint . Ramifications are much related to state constraints, which are properties of the world that are always true. These implicit effects can often be compiled into the action specification, i.e. all implicit effects are made explicit by adding them to the action definition. This will however make these definitions much longer. Examples of state constraints are the following:

$$\text{on}(X, Y, s) \rightarrow \neg \text{on}(Y, X, s) \quad \text{and} \quad \text{on}(X, Y, s) \wedge \text{on}(X, Z, s) \rightarrow Y = Z$$

These constraints enforce certain properties on descriptions of states of the world, but the use of these constraints when determining the effects of actions makes additional inference mechanisms necessary (see Chapter 6 for more detailed information on implementations). See (Reiter, 2001, Appendix B) for more details on state constraints and compiling implicit into effect axioms. Ramifications and state constraints have much in common with the definition of background knowledge predicates in LP and ILP, and more specifically to *saturation* (see Section 4.3.2.2) where the implicit knowledge (i.e. derivable background facts) can be computed from the explicit facts (i.e. an interpretation).

³⁵The work by Lin (2003) is mainly concerned with propositional STRIPS.

³⁶Though, some aspects of the SC appeal to second-order quantification (see Reiter, 2001).

4.4.3 Beyond Basic Action Theories

Basic action formalisms are sufficient for most methods in this book, as we are mainly interested in modeling and solving MDPs over relational domains. The state–the–art in commonsense reasoning (Reiter, 2001; Brachman and Levesque, 2004; Mueller, 2006) is much more advanced than the limited setting relational RL research has focused on so far. Still, many extensions that are only a little beyond basic formalizations are of interest.

One extension that is vital for relational RL is the incorporation of *uncertainty*. For example, there can be uncertainty about the world and the current state (e.g. as in POMDPs). More specifically, there can be uncertainty about the *outcome of actions*, as in MDPs. In the next sections we will introduce some simple probabilistic extensions of action formalisms that can be used for specifying RMDPs. Uncertainty has been investigated in planning systems, one of the closest areas to relational RL. Probabilistic action formalisms such as PPDDL (Younes and Littman, 2004) extend action definitions with multiple outcomes, subjected to a probability distribution. Many probabilistic planning approaches exist (Blythe, 1999) though uncertainty introduces a fundamental problem for the notion of a *plan*; execution of a plan can still fail on a concrete example because of probabilistic action outcomes. ML approaches can be used to overcome some difficulties in general planning (e.g. see Weld, 1999; Zimmerman and Kambhampati, 2003). A more general solution is to frame the problem into the (R)(MDP) context and focus on inducing *policies* instead of plans. Combinations of heuristic search planning and relational RL approaches is a useful way to make use of results in planning (see Section 2.5.2 and Karabaev and Skvortsova, 2005). Because of the added complexity of working with probabilities in first-order domains, many current planning approaches focus on efficient propositional approaches (see Littman, 1997; Russell and Norvig, 2003, Section 11.5).

There are many other extensions that have been investigated in planning systems research (see also Younes *et al.*, 2005, on the first international competition for probabilistic planning systems). For example, recent versions of PDDL (Ghallab *et al.*, 1998) support numeric functional fluents, explicit representation of time and duration and specification of plan metrics (rewards) as part of problem instances. Also the use of background knowledge predicates is supported (see the ramification problem in Section 4.4.1.2) (Edelkamp and Hoffman, 2004). Another recent extension of PPDDL supports general probability distributions for stochastic action effects (Teichteil-Koenigsbuch, 2008). Further extensions considered in current action formalisms are continuous actions, concurrent actions and events, continuous change, cumulative and canceling effects and many more (see Brachman and Levesque, 2004; Mueller, 2006, for pointers to the literature). All of these go far beyond the traditional MDP context. We point to the characteristics of logical formalisms as described in Section 4.2.2 and note here that for all of these extensions, computational complexity will increase. Additionally, most extensions introduce unsolved, conceptual problems when incorporated in the (R)(MDP) context and when combined with value functions and RL and DP algorithms (see also next sections).

Even more general extensions to action formalisms embed them into fully cognitive architectures. For example, planning and (relational) RL usually focus on learning a specific task or skill. These skills can be incorporated into behavior hierarchies (as in HRL, see Section 3.8). Extensions of planning to hierarchical task networks (HTN) have been investigated and the translation to hierarchical, relational RL is relatively straightforward (e.g. see Driessens and Blockeel, 2001; Aycenina, 2002; Roncagliolo and Tadepalli,

2004). However, embedding them into more general architectures such as used in agents (Wooldridge, 2002) and cognitive robotics (Reiter, 2001) is more complicated (see van Otterlo *et al.*, 2003, 2007, and in Chapter 7 we will investigate this topic further).

4.5. Learning Sequential Decision Making in Relational Domains

In the previous sections, we have encountered all the tools required for setting the stage for a framework for learning sequential decision making in stochastic, first-order domains. In this section we lift representations and learning algorithms to the first-order case. We start by lifting the problem representation (i.e. MDPs) in Section 4.5.1 and follow with lifted value functions and policies in Section 4.5.1.2. The algorithmic part will be obtained by lifting the PIAGET principle from Section 3.3.2 and defining new learning patterns based on logical deduction and induction mechanisms in Section 4.5.2. The section ends with a characterization of the five abstraction types from Chapter 3, now in relational domains and a definition of relational RL from four different viewpoints.

4.5.1 Lifting the MDP Framework to First-Order Domains

Solving sequential decision making problems under uncertainty is about algorithms that operate on suitable *representations* of the problem. In this section we *lift* all components of the standard MDP framework to the first-order case. This includes first-order representations of transition functions, reward functions, value functions and policies. Most importantly, we introduce FORMs, which are general, first-order specifications of RMDPs. Note that, unless stated otherwise, we focus on fully observable, finite RMDPs, to simplify matters and because most relational RL methods are based on these models.

In this section we focus on *representation*. We cannot do justice to the great variety of logical formalisms that have been and can be used for relational RL. We choose to formalize matters using a general form of FOL, without paying much detail to the algorithmic part of certain representations (e.g. theorem proving, model checking) nor to computational complexity or compactness of representations. Much of this, and pointers to relevant literature, has been described in the previous sections. Furthermore, many full or partial reductions and translations between formalisms exist. For a description of what it means to lift the MDP framework, general FOL suffices; for more detailed descriptions of individual formalisms we refer to Chapters 5 and 6. The term *abstract* – which will be used often – denotes any abstraction level that is more compact than the ground RMDP.

Additional general descriptions of learning sequential decision making in first-order domains can be found in (Boutilier, 2001; Kaelbling *et al.*, 2001; van Otterlo and Kersting, 2004; Younes and Littman, 2004; Tadepalli *et al.*, 2004; van Otterlo, 2005).

A Note on Terminology. The relative youth of relational RL makes that there is no consensus yet on a unified terminology for MDPs over first-order domains. Some authors use RMDP for the concrete instance of an MDP over some first-order language (e.g. Mausam and Weld, 2003) whereas others use it for a *template* (i.e. abstraction level) for multiple, similar RMDPs (e.g. Guestrin *et al.*, 2003a). Some others talk about the abstraction level, but distinguish between *relational* MDPs (restricted to implicitly existentially quantified formulas) and *first-order* MDPs (where explicit universal and existential quantifiers are allowed) (Sanner and Boutilier, 2006). Next, there are *logical* MDPs (Kersting and De

Raedt, 2004), *relationally factored* MDPs (Croonenborghs *et al.*, 2006b), and many other definitions of relational MDPs (e.g. see van Otterlo, 2004a; Roncagliolo and Tadepalli, 2004; van Otterlo, 2005; Gardiol and Kaelbling, 2006b). Several other formalizations exist in the *cognitive robotics* literature (e.g. see Boutilier *et al.*, 2000b) and *probabilistic planning* (Blythe, 1999; Younes and Littman, 2004; Younes *et al.*, 2005) that are based on the same semantics of a transition system over first-order structures.

In this book we use the following conventions. A *relational* MDP is a regular MDP where the states and actions consist³⁷ of ground, relational atoms, see Definition 4.1.1, much in line with Herbrand semantics. A *first-order* MDP (FOMDP) is a regular, possibly infinite, MDP where the states are first-order structures (i.e. a generalization of an RMDP), in line with Tarskian semantics. A *first-order represented* MDP (FORM) is a specification of a first-order logical, decision-theoretic problem that may *induce* typical *instances* of the problem (either first-order or relational), once instantiated with a set of objects.

4.5.1.1 REPRESENTING FIRST-ORDER AND RELATIONAL MDPs

The employment of logical representation languages allows for the exploitation of structure in e.g. states and actions. Recall from Definition 4.1.1 that an RMDP consists of a set of states S , i.e. relational interpretations, and a set of actions A , i.e. ground action atoms. The transition function T and reward function R operate on A and S . In this section we describe ways to represent S , A , T and R using FOL. We use a generic FOL language with no restrictions on expressivity or reasoning complexity. Additionally, we assume that any functor occurrence $f(X) = v$ has been replaced³⁸ by a relational atom $f(X, v)$.

Logical abstractions in the RMDP framework are typically used for three tasks: to compactly represent *sets* (e.g. sets of states), to represent a mapping from first-order structures to values (e.g. state value functions), and to represent symbolic mappings (e.g. from states to actions). Many of these abstractions will be *partitions* though this is by no means obligatory. We first introduce *multi-part abstractions* (MPA) which are useful devices for these representational tasks.

DEFINITION 4.5.1 ▶ Let Γ be a logical vocabulary and let \mathbb{A} be the set of all Γ -structures \mathcal{A} . A **multi-part abstraction (MPA) over \mathbb{A}** is a list $[\varphi_1, \dots, \varphi_n]$, where each φ_i ($i = 1 \dots n$) (called a **part**) is a formula. A structure $\mathcal{A} \in \mathbb{A}$ is covered by an MPA iff there exists a part φ_i ($i = 1 \dots n$) such that $\mathcal{A} \models \varphi_i$. An MPA is a **partition** iff for all structures there is exactly one part that covers it. Parts of non-partition MPAs are sometimes called **first-order features**. MPAs can be endowed with values resulting in the **value-MPA** $[\langle \varphi_1, v_1 \rangle, \dots, \langle \varphi_n, v_n \rangle]$, where each tuple $\langle \varphi_i, v_i \rangle$ ($i = 1 \dots n$) consists of a part φ_i and a value v_i . This is shorthand notation for:

$$t = [\langle \varphi_1, t_1 \rangle, \dots, \langle \varphi_n, t_n \rangle] \equiv \bigvee_{i \leq n} \{ \varphi_i \wedge t = t_i \}$$

An extension to **product-MPAs** defined over Cartesian products consists of parts $\langle \varphi_i, \psi_j \rangle$ where φ_i provides an abstraction of the first, and ψ_j an abstraction over the second set in the product space.

³⁷Thus, we consider Herbrand interpretations of a Datalog language.

³⁸Note that we assume that the set of values for $f(X)$ is finite. This is called flattening.

An MPA μ over Σ induces a set of equivalence classes over Σ , possibly forming a partition. In other words, μ is a compact representation of a first-order *abstraction level* over Σ . MPAs are to be seen as *sets*, though additional orderings can be imposed that render it a list. An element $\sigma \in \Sigma$ is covered by a part $\langle \varphi \rangle$ iff $\sigma \models \varphi$. The specific way in which it is computed (e.g. resolution, subsumption) is dependent on the logical system, and additionally on whether a background theory is supplied.

Operations can be defined on MPAs, for example to compute the sum of value-MPAs, or to compute the intersection of two partitioning MPAs. Logical *reduction*, or *simplification* operations can be defined on single MPAs to generate more compact syntactic descriptions (see Gottlob and Fermüller, 1993, for examples in clausal logic). An *MPA-operation* is a function that takes two MPAs μ_1 and μ_2 , and returns an MPA μ . For partitioning MPAs μ_1 and μ_2 , the intersection MPA $\mu = \mu_1 \cap \mu_2$ consists of all parts $\langle \varphi \wedge \psi \rangle$ where $\varphi \in \mu_1$ and $\psi \in \mu_2$. Operations on value-MPAs do arithmetic on value partitions. Let μ_3 and μ_4 be two value-MPAs over the same set. Operations include \otimes (multiplication) and \oplus (summation). These generate new MPAs consisting of $\langle \varphi_i \wedge \psi_j, t \cdot t' \rangle$, whenever $\langle \varphi_i, t_i \rangle \in \mu_3$ and $\langle \psi_j, t_j \rangle \in \mu_4$ (for \otimes) and $\langle \varphi_i \wedge \varphi_j, t_i + t_j \rangle$ (for \oplus).

MPAs introduce a *declarative representation* of partitions and first-order abstractions, though in most cases they will be represented more compactly using decision lists, trees or ADDs (see Section 4.2.3), which, in turn, will also determine how operations such as \cup , \otimes and \oplus will be implemented.

Abstract States and State Spaces. There are two types of states. The first are actual states of the environment, i.e. Herbrand interpretations, which we have encountered throughout this chapter (e.g. see Definition 4.1.1). The second type of state is called an *abstract state*, which is a logical sentence, specifying *properties* of states. For example, the abstract state $\forall X(\text{on}(X, \text{floor}))$ denotes the set of states in which all blocks are on the floor. An abstract state S represents *partial knowledge* about the actual state, and clusters a set of states into an equivalence class $\llbracket S \rrbracket$. As we deal only with fully observable worlds, this partial knowledge is only due to abstraction. Most relational RL systems limit the expressiveness of abstract states to (existentially quantified) *conjunctive* states (or, *queries*), which are states of the form $\exists \vec{X}(\text{p}_1(\vec{X}) \wedge \dots \wedge \text{p}_n(\vec{X}))$ where $\text{p}_1, \dots, \text{p}_n$ are predicates. Often the existential quantification is implicit and abstract states are represented by just the set of atoms. An abstract state S covers a ground state s iff $s \models S$, which can for conjunctive states be decided using e.g. θ -subsumption (e.g. Kersting and De Raedt, 2004) or resolution (e.g. van Otterlo, 2004a). For more expressive abstract states one may need a full-scale theorem prover (e.g. see Boutilier *et al.*, 2001, and further Section 4.2.1).

Abstract state spaces compactly specify in a logical way an RMDP's state space S as a set of abstract states, and can be defined as follows:

DEFINITION 4.5.2 \blacktriangleright An **abstract state space** is an MPA $[\varphi_1, \dots, \varphi_n]$, where each φ_i ($i = 1 \dots n$) is an abstract state. An **abstract state-action space** is an MPA $[\langle \varphi_1, \alpha_1 \rangle, \dots, \langle \varphi_n, \alpha_n \rangle]$, i.e. a product-MPA over the state-action space $S \times A$.

Only some systems use explicit representations of state (e.g. Morales, 2003) or state-action spaces (e.g. van Otterlo, 2004a), though most systems use these partitions to model state and state-action *value functions* (see further in this section). An additional class of systems does not represent the state(-action) space as a partition of logical formulas, but instead

transforms it using a set of first-order *features* (e.g. Walker *et al.*, 2004). Yet others represent it as a limited set of ground states (and apply instance-based learning) (e.g. Driessens and Ramon, 2003) or an *envelope* of ground states (Gardiol and Kaelbling, 2003). An additional class of representations uses *graph-based* formalisms (Gärtner *et al.*, 2003; Dabney and McGovern, 2006), though these can be mapped into query-based logical form.

Abstract Actions and Transition Functions. Transition functions for FOMDPs are usually based on action definitions such as described in Section 4.4, extended with *probabilistic outcomes*. The usual trick is to *decompose* a probabilistic action into a set of deterministic action outcomes under *nature's control*, i.e. once the probabilistic action is selected for execution, the actual result is influenced by the probability distribution over its deterministic outcomes. A general form of such a probabilistic operator is the following.

DEFINITION 4.5.3 ► A **probabilistic, first-order operator definition** is a tuple

$$\langle \text{pre}, \text{action}, [\langle \text{effects}_1, p_1 \rangle, \dots, \langle \text{effects}_n, p_n \rangle] \rangle$$

where both preconditions *pre* and effects effects_i ($i = 1 \dots n$) can be arbitrary formulas and *action* is an action atom with arguments. For the probabilities p_i ($i = 1 \dots n$) in the effect MPA it holds that $0 \leq p_i \leq 1$ and³⁹ $\sum_{i=1}^n p_i = 1$.

Although preconditions and effects can be arbitrary formulas, their expressiveness depends on the underlying action formalism. The general language PPDDL (Younes and Littman, 2004) is the probabilistic extension of PDDL which extends STRIPS (and ADL) in ways described in Section 4.4.2.1. An example of a (first-order) *probabilistic* STRIPS operator (Boutilier and Dearden, 1994; Kushmerick *et al.*, 1995) is the following⁴⁰ extension of the *move* action from Example 4.4.1.

$$\left\langle \{ \text{cl}(X), \text{cl}(Y), \text{on}(X, Z) \}, \text{move}(X, Y) \right. \\ \left. [\langle \text{add} : \{ \text{cl}(Z), \text{on}(X, Y) \}, \text{del} : \{ \text{cl}(Y), \text{on}(X, Z) \}, 0.9 \rangle, \langle \text{add} : \emptyset, \text{del} : \emptyset, 0.1 \rangle] \right\rangle \quad (4.7)$$

This tells us that with probability 0.9 the action will have its standard effects, though with probability 0.1 there will be no effects. We assume that the effects of the action are mutually exclusive. Otherwise the probability distribution is ill-defined and additional methods⁴¹ are required to compute the probability of outcomes (see Gardiol, 2003). In addition, one usually assumes that variables used in the action term, appear in the precondition and the effects, and furthermore no other free variables.

³⁹Zettlemoyer *et al.* (2005) extend this definition with a so-called *noise* outcome, which can take up some probability mass but is left unspecified. This is useful for rarely occurring effects that are too hard to model exactly, like the effects of knocking over a complete tower of blocks and predicting where all blocks will be on the floor after this action.

⁴⁰Note that we omit here inequality atoms such as $X \neq Y$, but we assume it holds for all variables.

⁴¹Gardiol (2003) uses the *noisy-OR* rule, which is a simple way of describing the probability for an effect given the presence or absence of its causes. It makes a very strong assumption that each causal process is independent of the other causes: the presence of any single cause is sufficient to cause the effect, and the absence of the effect is due to the independent "failure" of all causes.

Stochastic actions in the SC are defined in a similar way (Boutilier *et al.*, 2001; Reiter, 2001), though they require a complete axiomatization of the stochastic action and its deterministic components. The failing `move` action in our example would require the definition of a standard `moveSucc` action (defining the first, deterministic outcome) and a `moveFail` action (representing the second, deterministic outcome). Probabilities for outcomes are then specified as additional axioms saying that

$$\begin{aligned} \text{prob}(\text{move}(X, Y, s), \text{moveSucc}(X, Y, s)) &= 0.9, \text{ and} \\ \text{prob}(\text{move}(X, Y, s), \text{moveFail}(X, Y, s)) &= 0.1 \end{aligned}$$

Additional conditions can be placed on the context in which the stochastic action has these effects, i.e. using condition $\rightarrow (\text{prob}(a, a_i) = p_i)$, where a is the stochastic action and a_i are its deterministic outcomes. Other action logics, such as the FC and EC (see Section 4.4.2), can be extended with probabilistic actions in similar ways.

Similar to propositionally factored MDPs (see Section 3.5), first-order action dynamics can be represented more compactly. In fact, many of the SRL techniques from Section 4.3.3 would be amenable for application in relational RL contexts. Among the few approaches that have considered this direction, Guestrin *et al.* (2003a) use PRMs to represent probabilistic actions in factored form, though use them mainly as a structural *dependency structure* to compute parameters such as state values and probabilities. Croonenborghs *et al.* (2006a) use *probability trees* to capture *partial models* of the transition dynamics, i.e. they estimate probabilistic effects of individual atoms and combine these probabilities when applying a forward planning algorithm. Earlier work by Gardiol (2003) reports on similar, partial models. Both have to consider additional *conflict resolution* techniques when multiple rules apply simultaneously. Joshi *et al.* (2006) go furthest in obtaining compactness of the representation of probabilistic actions. They use first-order ADDs to represent action dynamics (see Section 4.2.3.1 and Chapter 6 for further discussion).

Abstract Goals and Reward Functions. *Goals* can be specific states (i.e. interpretations) or sets of states, as commonly used in planning systems. For the (FO)MDP context we will look at the more general goal specification in terms of a reward function definition in terms of abstract states (see also Section 2.4 on this matter).

DEFINITION 4.5.4 ▶ An **abstract state**⁴² **reward function** \mathcal{R} maps states $s \in S$ to real-valued numbers, denoted as the partitioning MPA $[\langle \varphi_1, r_1 \rangle, \dots, \langle \varphi_n, r_n \rangle]$ over S where $\varphi_1, \dots, \varphi_n$ are abstract states, and r_1, \dots, r_n are reals. Each $\langle \varphi_i, r_i \rangle$ ($i = 1 \dots n$) assigns the value r_i to all states s in the set $\llbracket \varphi_i \rrbracket$, denoted $\mathcal{R}(s) = r_i$.

An *abstract goal* φ can be translated into an abstract reward function $[\langle \varphi, 1 \rangle, \langle \neg\varphi, 0 \rangle]$, but there are many other possibilities (Koenig and Liu, 2002). In addition to a reward function, some systems use a separate *cost function* for actions (e.g Karabaev *et al.*, 2006). If the task is episodic, goal states are made absorbing, by adjusting the action definitions accordingly, by adding artificial *noop* actions (see Chapter 6) that loop into goal states, or by a separate specification.

Several methods use different sets of predicates to distinguish between those that describe the world and those that describe the goals (Kharon, 1999b; Martin and Geffner,

⁴²We will only define state value functions here. State-action value functions or state-action-state value functions can be defined in similar ways, see Chapter 2.

2000; Yoon *et al.*, 2002; Fern *et al.*, 2006). Actions will then change the world predicates in the usual way, but leave the goal predicates untouched. Džeroski *et al.* (1998) (and extensions, see Driessens, 2004) utilize goal atoms in the representation of value functions in order to generalize over concrete objects that might be mentioned in the goals. Declarative goals such as these can also help in transferring learned value functions and policies across domains (Driessens *et al.*, 2006a).

Some work has considered *additive reward structures* in relational RL (Guestrin *et al.*, 2003a; Sanner and Boutilier, 2005; Asgharbeygi *et al.*, 2006). An additive reward function decomposes the reward into a number of contributions and can be represented as a (non-partitioning) MPA. Asgharbeygi *et al.* (2006) attach reward to *conceptual predicates* whereas Guestrin *et al.* (2003a) assign rewards to objects satisfying some condition (e.g. when a block is at the right place). Additive reward function definitions make it easier to specify (declaratively) individual reward contributions, though they make (value-based) learning algorithms more complex.

An important aspect of abstract reward functions is the expressivity of the goals on which they are based. Some systems assume fully specified, ground goal states (e.g. Fern *et al.*, 2006), others use abstract states as goals (e.g. Kersting *et al.*, 2004) and yet others force the system not to use constants in abstract goals (e.g. Driessens, 2004). The learning setting (i.e. model-based, or model-free) is important for how goals can be represented. In the model-free setting, goals merely represent *tests* on the current state, such that they can be represented by arbitrarily complex formulas or programs. The model-based setting, where the algorithm might have to reason symbolically about the goal, puts more restrictions on the representation of goals and abstract reward functions. Consider a reward that depends on the number of domain objects that satisfy some criterion, e.g. the number of boxes delivered in a logistics domain. If the reward depends on *all* boxes being there, one would represent this as $\forall X(\text{box}(X) \wedge \text{here}(X))$. This requires knowledge about the concrete domain when reasoning explicitly about the reward structure. More importantly, it cannot be represented by query-based formalisms, such as STRIPS-like systems and Horn logic. Handling universally quantified reward functions effectively is one of the most pressing issues in first-order domains (see also Gretton and Thiébaux, 2004a; Sanner and Boutilier, 2006). One solution was provided by Sanner and Boutilier (2006) who use an additive reward function decomposition (see more in Chapter 6). Most relational RL systems use restricted formalisms and are only capable of representing abstract states and reward functions as queries.

Logical Representations of First-Order MDPs. Abstract states, actions and reward functions can now be used to define abstract specifications of RMDPs. Specifications of first-order MDPs typically borrow a logical language from the field of FOL, action definitions from the field of planning, and one adds probabilistic outcomes to these actions and specifies a reward function. Here we formalize a logical representation of a finite RMDP. We assume that we have access to some logical language Γ that is rich enough to provide us with a syntactic machinery for predicates, constants, functions, variables, connectives and action names.

DEFINITION 4.5.5 ▶ A **First-Order Represented MDP (FORM)** is a tuple $\langle \mathcal{D}, \mathcal{P}, \mathcal{A}, \mathcal{R}, \mathcal{B} \rangle$ where \mathcal{D} is a finite set of objects, \mathcal{P} is a finite set of predicates, \mathcal{B} is a finite background theory, \mathcal{A} is a finite set of probabilistic actions, and \mathcal{R} is an abstract reward function. Both

\mathcal{A} and \mathcal{R} are based on $\mathcal{D} \cup \mathcal{P} \cup \mathcal{B}$.

Similar to the action formalisms in Section 4.4, we can distinguish between modeling and axiomatizing FORMS. For example, for STRIPS or ADL based languages, FORMS are useful models for learning and planning in concrete problems. When the FORM is defined in an action logic such as SC or FC they support logical *reasoning* over properties of the system, because all components are fully described as logical axioms. Nevertheless, regardless of the logical language, FORMS are a core model of current relational RL formalizations. Let us consider an example using STRIPS.

EXAMPLE 4.5.1 ▶ Let $\mathcal{F} = \langle \mathcal{D}, \mathcal{P}, \mathcal{A}, \mathcal{R}, \mathcal{B} \rangle$ be a FORM, where

$$\mathcal{D} = \{\text{a, b, c, d, floor}\},$$

$$\mathcal{P} = \{\text{on}/2, \text{clear}/1\}$$

\mathcal{A} is given by Equation 4.7

$$\mathcal{R} = [\langle \forall X \text{on}(X, \text{floor}), 10 \rangle, \langle \neg \forall X \text{on}(X, \text{floor}), 0 \rangle]$$

$$\mathcal{B} = \emptyset.$$

\mathcal{F} specifies a probabilistic BLOCKS WORLD with four blocks in which *move*/2 is the only possible action. The application of this action fails with probability 0.1. All states⁴³ in which all blocks are on the floor, emit a reward 10 and all other states have reward 0. Goal states can be made absorbing by modifying the transition function accordingly.

Several similar, explicit definitions of FORMS have appeared in the literature that share this common structure (e.g. see Mausam and Weld, 2003; Guestrin *et al.*, 2003a; van Otterlo, 2004a; Roncagliolo and Tadepalli, 2004; Kersting and De Raedt, 2004; Croonenborghs *et al.*, 2006b; Gardiol and Kaelbling, 2006b; Fern *et al.*, 2006). All are based on the idea that objects, together with predicate and action *templates* make up the state and action spaces. Differences between approaches are superficial or reside in small modifications such as the addition of *types* (Mausam and Weld, 2003; Gardiol and Kaelbling, 2006b) or an *initial state distribution* (Fern *et al.*, 2006). First-order representations of RMDPs have been proposed in other action formalisms (Bacchus *et al.*, 1999; Boutilier *et al.*, 2001; Großmann *et al.*, 2002; Sanner and Boutilier, 2006), stochastic modeling languages such as IBAL (Pfeffer, 2001), in restricted settings such as *functional* STRIPS (Geffner and Bonet, 1998; Geffner, 2000) and other action languages (Poole, 1997a; Geffner and Bonet, 1998). Furthermore, recent advances in PDDL introduce similar formalizations (Younes and Littman, 2004; Younes *et al.*, 2005) that have the same kind of stochastic transition systems (i.e. (R)MDPs) as their semantics.

Background Theory. The background theory \mathcal{B} in the FORM definition serves a number of purposes, but two of the most important are the extension of the state language with additional predicate definitions (i.e. as derived relations in planning (Edelkamp and Hoffman, 2004) or background predicates as in ILP) and the provision of domain constraints and ramifications (i.e. the static and dynamic laws in the domain). For example, a typical BLOCKS WORLD state is described in terms of *on* and *clear*, though an abstract state might use the *derived* predicate *onTop*. These derived predicates might also be used in the action definitions and abstract reward functions. For example, the Q-RRL system (Džeroski *et al.*, 2001a) uses background knowledge for the induction of logical trees that represent

⁴³Note that there is only one state with such a reward for a fixed domain size.

Q -functions. Furthermore, \mathcal{B} might specify constraints such as $\forall XY(\text{on}(X, Y) \rightarrow X \neq Y)$, saying that if X and Y are on top of each other, they should be different objects. An explicit usage of such constraints will be described in Chapter 6. All in all, the background theory provides means to cope with the characteristic problems of action languages described in Section 4.4.1.2

RMDPs Provide Semantics to FORMs. FORMs are representational devices to compactly specify a first-order or relational MDP, though in the literature this correspondence is often tacitly assumed to exist via the semantics of the logical system that is used. In the following theorem we provide a formal connection between FORMs (Definition 4.5.5) and RMDPs (Definition 4.1.1). See for related argumentations (Kersting and De Raedt, 2004; Gardiol and Kaelbling, 2006b).

THEOREM 4.5.1 \blacktriangleright Every FORM $\mathcal{F} = \langle \mathcal{D}, \mathcal{P}, \mathcal{A}, \mathcal{R}, \mathcal{B} \rangle$ specifies a finite RMDP $M(\mathcal{F}) = \langle S, A, T, R \rangle$ as its intended model.

Proof. (informal outline)

A complete proof would demand a full specification of the logical system in which the FORM is formalized. Regardless of the language, however, a number of steps provide a general, constructive proof. The crucial point is to see a FORM as a syntactic, logical specification, where the underlying RMDP serves as its semantics.

First, the set of states S is formed by constructing the set of first-order interpretations over \mathcal{P} , \mathcal{D} and \mathcal{B} . In a logic programming context this amounts to constructing all Herbrand interpretations from the Herbrand base $\text{HB}(\mathcal{P} \cup \mathcal{D} \cup \mathcal{B})$. From the finiteness of \mathcal{P} , \mathcal{D} and \mathcal{B} , it follows that $|S|$ is finite. A finite set of ground action atoms A is constructed from all action names occurring in \mathcal{A} combined with \mathcal{D} . We first set all transition probabilities $T(s, a, s')$ to zero ($s, s' \in S, a \in A$).

Second, for any state $s \in S$ we construct the set of applicable actions $A(s)$ from \mathcal{A} . An action definition \mathcal{A} is specified as $\langle \text{pre}, \text{action}, \lfloor \langle \text{effects}_1, p_1 \rangle, \dots, \langle \text{effects}_n, p_n \rangle \rfloor \rangle$ (see Definition 4.5.3). From $s \models \text{pre}\theta$, where θ is a variable substitution, we obtain all available ground actions $\text{action}\theta$. For each action $\text{action}\theta$ and for all $i = 1 \dots n$ we compute all states s' such that applying all effects $\text{effects}_i\theta$ in s (keeping in mind that all state constraints and ramifications in \mathcal{B} are enforced) yields s' and add p_i to the transition probability $(s, \text{action}\theta, s')$. Finally, the probability for all transitions (s, a, s') is normalized over all transitions (s, a, s'') , where $s'' \in S$.

Third, the reward function definition \mathcal{R} is used to assign rewards to individual states using $R(s) = r$ if $\langle \varphi, r \rangle \in \mathcal{R}$ and $s \models \varphi$. A goal state $s \in S$ is made absorbing by specifying all transition probabilities $T(s, a, s) = 1$ for all $a \in A(s)$. □

Domains, Instances and Families. A typical distinction made by planning formalisms such as PDDL is that between a *domain description*, that describes general characteristics of some domain, and *instances* that are concrete problems in that domain (see Section 4.4.2.1). In spirit of this distinction one can define three levels of description in the context of FORMs. Let \mathcal{F} be the FORM $\langle \mathcal{D}, \mathcal{P}, \mathcal{A}, \mathcal{R}, \mathcal{B} \rangle$.

- **Domain description.** A domain is described by a language for specifying states and actions, a description of how actions operate, and possibly a background theory,

i.e. $\langle \mathcal{P}, \mathcal{A}, \mathcal{B} \rangle$. This is similar to standard planning domain descriptions, with the exception that actions are now probabilistic and we supply a separate, additional background theory.

- **Problem description.** When an abstract reward function definition is added to the domain description, we obtain $\langle \mathcal{P}, \mathcal{A}, \mathcal{R}, \mathcal{B} \rangle$, describing the behavior of a stochastic system with rewards. It describes a specific problem template because it describes the dynamics of the world as well as which (abstract) states are desirable.
- **Instance.** Only when adding a set of domain objects \mathcal{D} to a problem description, we fix the number of states and actions, and with that, the size of the problem. Theorem 4.5.1 shows that an instance corresponds to a fully described RMDP.

Recalling Example 4.5.1, we see that the domain description specifies that states are described in terms of `on/2` and `clear/1` and how the `move/2` action is defined. A specific problem is introduced when it is specified that the goal is to put all blocks on the floor ($\forall X \text{on}(X, \text{floor})$). Once we specify a concrete set of objects $\{a, b, c, d, \text{floor}\}$ we obtain a fully-specified RMDP. Additionally, especially in (episodic) model-free contexts, one can define an *initial state distribution* that delivers *starting states* according to some, (domain-specific) distribution (see also Chapter 5). In Chapter 6 we will introduce *Markov decision programs* which are simple instantiations of FORMs and in Chapter 5 we discuss a domain-specific initial state distribution for BLOCKS WORLD (see also Section 4.1.2).

DEFINITION 4.5.6 ▶ A problem description $\langle \mathcal{P}, \mathcal{A}, \mathcal{R}, \mathcal{B} \rangle$ generates a range of FORMs $\langle \mathcal{D}, \mathcal{P}, \mathcal{A}, \mathcal{R}, \mathcal{B} \rangle$, each differing in the domain instantiation \mathcal{D} . All RMDPs modeled by these FORMs together form a **family**⁴⁴ of RMDPs.

For instance, take the FORM in Definition 4.5.5 where \mathcal{D} is left unspecified. The remaining specification contains all structural parts of the domain and the problem. An optimal policy for this domain would not depend on the number of blocks, e.g. it would just break all towers by placing all blocks on the floor. Furthermore, FORMs support *parameterization* of goals and policies. For example, the goal $\exists X, Y \text{on}(X, Y)$ contains two variables that are left unspecified, but can be used in a policy such that when the actual goal is changed from `on(a, b)` to `on(b, c)` the policy can change accordingly⁴⁵. It turns out that algorithms can be developed that operate at the level of families instead of instances, and that solutions can be obtained that apply to *any* instance in the problem domain (see also Section 4.5.2.2). The shared structure in all instances in a problem domain offers opportunities for learning. In the words of Baum (2004, Sec. 11.3.1), “The BLOCKS WORLD class has a compact description and in addition its instances have a lot in common, so its instances are solvable”.

The concept of families of RMDPs, the ability to parameterize policies, value functions and reward definitions, and additionally the use of domain knowledge that is common to all instances in a problem domain, represent an important distinction with simpler representational strategies such as atomic and propositional (e.g. factored) representations.

⁴⁴Although we already encountered the problem, the term ‘families of RMDPs’ was coined by Roni Khardon during his talk at Dagstuhl 2005.

⁴⁵Note that in that case we need additional machinery to grant the policy access to these variables. For example, Džeroski *et al.* (2001a) put the goal in the root node of induced value functions and policies, whereas Fern *et al.* (2007) mark goal atoms with additional tags.

Extensions of FORMs. Finite RMDPs and their logical representation are relatively simple. Most representations in relational RL are based on standard action logics such as STRIPS and clausal logic. In Sections 4.2 to 4.4 we have described a variety of systems that might be used for relational RL. Richer action formalisms (see Section 4.4.3) and richer logical languages for reasoning and learning can – in principle – all be used to scale up the work in relational RL, for example to deal with probabilistic belief states, temporal reasoning, and richer domain models. In Chapter 7 we will describe richer agent models where RL is seen as one of many skills an agent possesses. Richer agent models can provide so-called *program constraints*, similar to task structures as used in hierarchical RL (see Section 3.8). Furthermore, multi-agent decompositions and concurrent actions (e.g. see Finzi and Lukasiewicz, 2004b) are also possible.

As for language extensions that would be useful in relational RL, there are at least three directions. First, *functors*, and *count* and *sum aggregators* are a topic of research in ILP methods (e.g. see Džeroski and Lavrac, 2001b) and for relational RL they are very relevant. For example, one would like to incorporate a functional fluent `numberOfItems` that counts how many items the agent is carrying, for example in the STARCRAFT domain (see Figure 4.1a). The amount should be updated when picking up or dropping items, and it could influence the outcomes of actions (e.g. there could be a limit on the amount of items the agent can carry) and it could influence the reward function (e.g. the agent gets rewarded for picking up many items). For instance, Guestrin *et al.* (2003a) uses count aggregators for this purpose. Second, full-scale handling of (real-valued) quantities would be useful and could, in turn, greatly influence the state value function. Third, dealing efficiently with *topological structure* is an important topic (see Dai and Goldsmith, 2007, for related ideas). Many (planning) domains contain topological structure, involving locations and relative distances between locations. Although these can be represented in first-order-logic, it would be much better to assume the location structure is fixed and use specialized (graph-based) algorithms for this part (see also Lane and Wilson, 2005)

4.5.1.2 ABSTRACT VALUE FUNCTIONS AND POLICIES

The first-order notions defined in the previous section supply representational machinery for posing the *problem* compactly, i.e. an RMDP. Solving it means computing a *policy*, possibly aided by value functions. Both are defined relative to some FORM, and are usually based on the same syntax. Abstract policies may apply to all instances of a family of RMDPs, whereas value functions usually apply only to a specific RMDP.

Abstract Policies. The goal of learning in sequential decision making is a policy (i.e. a kind of universal plan). Here, we focus on *reactive* policies for simple RMDPs and limited background theories. A first-order policy for an RMDP $M = \langle S, A, T, R \rangle$ is $\pi : S \rightarrow A$, i.e. a mapping from first-order interpretations, to actions. A general form is the following:

DEFINITION 4.5.7 ► An **abstract policy** $\tilde{\Pi}$ is an MPA $[\langle \varphi_1(\vec{X}_1), \alpha_1(\vec{X}_1) \rangle, \dots, \langle \varphi_n(\vec{X}_n), \alpha_n(\vec{X}_n) \rangle]$ where $\varphi_1(\vec{X}_1), \dots, \varphi_n(\vec{X}_n)$ are abstract states which form a partition of S , and each $\alpha_i(\vec{X})$ ($i = 1 \dots n$) is an action that shares variables with $\varphi_i(\vec{X})$. Each tuple $\langle \varphi(\vec{X}), \alpha(\vec{X}) \rangle$ forms a **policy rule** $\varphi(\vec{X}) \Rightarrow \alpha(\vec{X})$ interpreted as *if the current state is covered by $\varphi(\vec{X})$ then apply action $\alpha(\vec{X})$.*

Application of a policy rule $\varphi(\vec{X}) \Rightarrow \alpha(\vec{X})$ in a concrete state $s \in S$ is done by inferring

$s \vdash \varphi_i(\vec{X})\theta$, for some i . Then, the ground action $a = \alpha_i(\vec{X})\theta$ is executed, i.e. $\tilde{\Pi}(s) = a$. Note that there can be multiple ground actions a , due to multiple substitutions θ . This means that an additional decision is needed (usually random) on which concrete action to apply, and that the policy is basically *non-deterministic* in the ground RMDP. Consider the (partial) ground state $s_1 \equiv \{\text{on}(a, b), \text{on}(b, \text{floor}), \text{clear}(a), \text{on}(c, \text{floor}), \text{clear}(c)\}$ and the policy rule $\exists X, Y. \text{clear}(X), \text{clear}(Y) \Rightarrow \text{move}(X, Y)$. Both actions $\text{move}(a, c)$ and $\text{move}(c, a)$ can be applied, i.e. $\tilde{\Pi}(s_1) = \{\text{move}(a, c), \text{move}(c, a)\}$. This causes the policy to be non-deterministic⁴⁶ in the ground RMDP. Abstract policies that apply to all instances in some problem domain, are called *generalized policies* (Martin and Geffner, 2000, 2004).

Muller and van Otterlo (2005) use a *probabilistic interpretation* of a policy, in which the MPA representing the policy is non-partitioning, i.e. in a specific ground state s , multiple policy rules $\varphi(\vec{X}) \Rightarrow \alpha(\vec{X})$ may exist that cover s . The probability that a particular rule is chosen to generate an action is proportional to the relative number of groundings each rule provides. A similar technique, based on rule *fitness*, was used by Mellor (2005a).

An important distinction in policy representations is that between *generative* and *descriptive* representations. A generative representation is a template that will generate all valid actions for a given state s . An example is given in the previous paragraph where the action $\text{move}(X, Y)$ will only generate valid ground actions in s_1 , due to the template given by the abstract state. A descriptive policy rule such as $\text{clear}(X) \Rightarrow \text{move}(X, Y)$ applied to s_1 will cover the action $\text{move}(a, b)$ too, although it is not a valid action. Descriptive representations can only apply a *check*, assuming the list of all valid actions is supplied from outside. This is often assumed for model-free algorithms, where the agent gets a list of all valid actions to choose from for each state.

Abstract Value functions and Bellman Equations. Recalling from Chapter 2, a (state) value function represents for each state the expected future reward obtained by the agent while following some policy, under some performance criterion. Abstract value function representations are similar to abstract reward function representations, though their semantics is quite different.

DEFINITION 4.5.8 ▶ An **abstract value function (ASVF)** \tilde{V} maps states to real values, denoted by the MPA $[\langle \varphi_1, v_1 \rangle, \dots, \langle \varphi_n, v_n \rangle]$, where $\varphi_1, \dots, \varphi_n$ are abstract states and each $v_i \in \mathbb{R}$ ($i = 1 \dots n$). For every state $s \in S$, $\tilde{V}(s) = v_i$ if $s \in \llbracket \varphi_i \rrbracket$ for some $i \in [1 \dots n]$.

We assume that $v_i \neq v_j$ whenever $i \neq j$ ($i, j = 1 \dots n$). Otherwise, $\langle \varphi_i, v \rangle$ and $\langle \varphi_j, v \rangle$ would have been combined into $\langle \varphi_i \wedge \varphi_j, v \rangle$. For example, the ASVF $[\langle \text{clear}(a), 10 \rangle, \langle \neg \text{clear}(a), 0 \rangle]$ assigns a value 10 to all states in which a is clear and 0 to all other states. There is a significant difference between state value and state-action value functions (in the relational setting) due to dependencies between states and actions and variables bindings.

DEFINITION 4.5.9 ▶ An **abstract state-action value function (ASAVF)** \tilde{Q} can be represented using an MPA $[\langle \varphi_1(\vec{X}_1), \alpha_1(\vec{X}_1), q_1 \rangle, \dots, \langle \varphi_n(\vec{X}_n), \alpha_n(\vec{X}_n), q_n \rangle]$, similar to a policy MPA, augmented with a value for each abstract state-action pair. For every state-action pair $\langle s, a \rangle$ ($s \in S$ and $a \in A$), $\tilde{Q}(s, a) = q_i$ if $\langle s, a \rangle \in \llbracket \varphi_i(\vec{X}_i), \alpha_i(\vec{X}_i) \rrbracket$ for some $i \in [1 \dots n]$.

⁴⁶Note that we can always enforce the policy to be deterministic by posing an ordering on the domain objects and select between actions based on this ordering.

The language used in ASVFs and ASAVFs can be extended using a background theory. In contrast to abstract policies, abstract value functions generally do not apply to all instances in a domain. Each instance has its own (ground) value function. The *range* of values in this value function grows with the domain size (i.e. the state space size of the instance), and can become infinite when the underlying RMDP has an infinite (or even indefinite, see Kersting *et al.*, 2004) domain (see also discussions in Yoon *et al.*, 2002; Gretton and Thiébaux, 2004a). Several model-free relational RL algorithms (see Chapter 5) represent only ASAVFs and derive (policy) decisions online from them (e.g. Džeroski *et al.*, 2001a).

Both ASVFs and ASAVFs can be defined relative to an abstract policy $\tilde{\Pi}$, such that $\mathcal{V}^{\tilde{\Pi}}$ and $\mathcal{Q}^{\tilde{\Pi}}$ denote value functions when actions are selected according to policy $\tilde{\Pi}$. Furthermore, $\tilde{\mathcal{V}}^*$ and $\tilde{\mathcal{Q}}^*$ denote optimal abstract value functions.

THEOREM 4.5.2 ▶ For any finite FORM $\mathcal{F} = \langle \mathcal{D}, \mathcal{P}, \mathcal{A}, \mathcal{R}, \mathcal{B} \rangle$ there exist an optimal value function \tilde{V}^* and an optimal policy $\tilde{\Pi}^*$.

Proof. From Theorem 4.5.1 it follows that \mathcal{F} induces a finite RMDP $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$. Because M is a finite MDP, a unique optimal value function V^* and at least one optimal policy π^* exist for M . Set $\tilde{V}^*(s) = V^*(s)$ and $\tilde{\Pi}^*(s) = \pi^*(s)$ for all $s \in \mathcal{S}$. The existence of \tilde{V}^* and $\tilde{\Pi}^*$ is ensured by the fact that \tilde{V} and $\tilde{\Pi}$ can both represent the ground value function v and policy π respectively. (see also Ramon, 2005a,b). Whether \tilde{V}^* and $\tilde{\Pi}^*$ can be represented more compactly depends on the expressivity of the language used. \square

The Bellman equations for $\tilde{\mathcal{V}}$ (see Equation 2.3 in Section 2.3) are only defined in the ground RMDP, as a functional relation between the values of pairs of ground states. Lifted versions of these equations depend on the abstraction level, as we have seen in the previous chapter, more specifically in Section 3.4 on state aggregation (see also Li *et al.*, 2006). For example, equations can be defined when (soft) state aggregation is used (Singh *et al.*, 1995) and convergence of RL algorithms can be stated over a set of aggregate states. Such results can be immediately carried over to the relational case when only state abstraction is used (see Kersting and De Raedt, 2004). However, in a general relational setting there are some complications (e.g. see van Otterlo, 2004a, and the next chapter). Abstract actions provide an abstraction over the action space too, and usually they are dependent on the syntactic state or precondition representation through variables. This means that Bellman equations should be defined over abstract action specifications. Furthermore, abstract actions are usually specified in terms of preconditions and effects, quite independently of an abstract state space (if provided anyway). Still, Bellman equations can be defined in terms of structured representations of actions, forming the basis for *structured Bellman backups* in structured algorithms for factored MDPs (see Section 3.5). These structured Bellman backup operators and structured solution algorithms can be entirely lifted to a first-order level, and we deal with this topic extensively in Chapter 6.

Another way to estimate the value function $\tilde{\mathcal{V}}^{\tilde{\Pi}}$ for a given policy $\tilde{\Pi}$ is by sampling the underlying RMDP according to the policy (e.g. see Lecoeuche, 2001; Muller and van Otterlo, 2005), for example using policy rollout (Fern *et al.*, 2006). Value functions for policies displaying complex structure such as in GOLOG can be computed using forward application of this policy in off-line (Boutilier *et al.*, 2000b) and on-line (Soutchanski, 2001) settings. Bellman equations for stochastic programs were given by (McAllester, 1999, 2000; Soutchanski, 2001), resembling value function decompositions such as used

in hierarchical RL (see Section 3.8). See Chapter 7 for more on these matters.

So far, we have described *exact* value functions. For many complex domains, it will be hard to represent (or learn) exact representations of value functions for relational domains (Kersting *et al.*, 2004). One alternative is the *instance-based* representation used in RIB (Driessens and Ramon, 2003), that stores a set of ground instances of state-action pairs with their corresponding values. This set is updated by adding and deleting ground states, and values of these states are updated during learning. TRENDI (Driessens and Džeroski, 2005) combines this representation with a top-level ASAVF that first partitions the state-action space, where the concrete value for partitions is stored in an instance-based fashion. Related approaches aim at inducing ASAVF (and abstract policies) by representing (and learning) value functions at a ground level (Lecoeuche, 2001; Mausam and Weld, 2003; Cocora *et al.*, 2006).

Another alternative for exact ASVRs is to use an approximate representation based on *first-order basis functions* (or: *first-order features*), very similar to the linear value function decompositions described in Section 3.6.2.1.

DEFINITION 4.5.10 ► A decomposition into basis functions can be represented by the MPA $[\langle \varphi_1, w_1 \rangle, \dots, \langle \varphi_n, w_n \rangle]$ where each w_i ($i = 1 \dots n$) is a *weight* and the features can overlap. Each φ_i ($i = 1 \dots n$) acts as a binary feature (mapping to $[0, 1]$) such that the value of a state is computed as $\sum_{i=1}^n w_i \cdot \varphi_i$.

The number of basis functions is typically much smaller than the number of abstract states needed for exact value functions. Basis functions have been used in both model-free relational RL (e.g Walker *et al.*, 2004; Asgharbeygi *et al.*, 2006) and model-based relational RL (e.g. Sanner and Boutilier, 2005). Sanner (2005, 2006a) uses a naive Bayes representation of the *probability of reaching the goal state* (which for goal-based reward functions is equivalent to the state value function) based on a set of first-order features.

4.5.2 The PIAGET-Principle in First-Order Domains

In Section 3.3.2 we have outlined the core mechanisms of abstraction in the context of MDP solution algorithms, conceptualized in the PIAGET principle. We have formulated this principle independently of a particular representation and as a consequence, it also applies in the relational setting introduced in this chapter. In order to obtain this result, it was necessary to lift three necessary parts to the first-order case. First, in Section 4.2 we have described logical representation languages to describe relational worlds and to form first-order abstraction levels over all components in the MDP framework. Second, we have dealt extensively in Section 4.3 with mechanisms for the generation (e.g. induction) of first-order abstraction levels. Third, Section 4.4 described action formalisms for first-order domains. These three components enable again four PIAGET-levels to be distinguished for relational RL algorithms, in exactly the same fashion as in Chapter 3. Furthermore, in Section 4.5.3 we will discern five abstraction types in relational RL algorithms in exactly the same fashion as the previous chapter.

Lifting the representational and algorithmic aspects to first-order has resulted in the PIAGET-principle still being applicable in this new context, but it also introduces an important new aspect. Due to the fact that FORMS can be used to induce families of RMDPs and also because value functions and policies can generalize over many RMDPs, solution

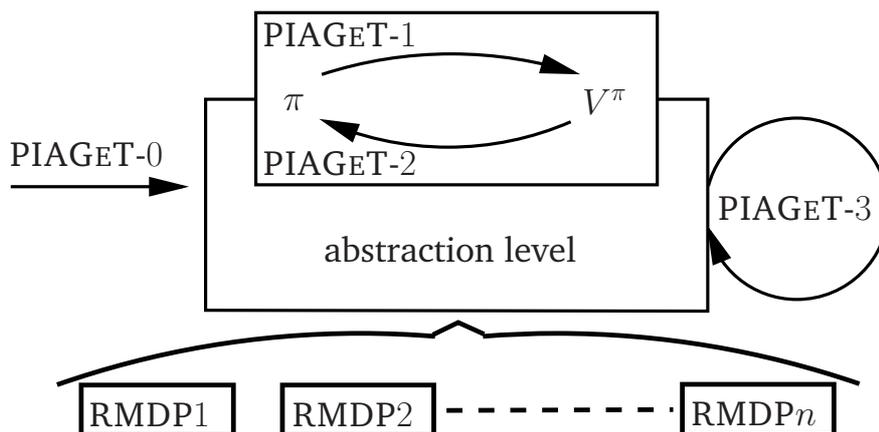


Figure 4.11: Policy Iteration using Abstraction and Generalization Techniques (PIAGET) in the context of RMDPs.

algorithms may generalize over families of RMDPs too. If we compare Figure 4.11 to Figure 3.8 in Chapter 3 we see that in the first-order setting the abstraction level at which learning and computation happens, generalizes over multiple RMDPs. This has a huge influence on the possibilities of learning algorithms and for *transfer* of learned solutions to other, similar problems. It also brings up a question on how value-based methods can be applied when the values they learn stem from multiple RMDPs. In addition, it also introduces possibilities for learning from multiple problem instances simultaneously, or quoting Baum (2004, p. 154): *“To extract structure, one must look at classes of problems”*.

The symbolic nature of first-order representations of the world has a strong influence on the type of algorithms that we will encounter. Most of the algorithms belong to PIAGET-3, i.e. these algorithms will have to combine learning of parameters and (logical) structures. The compactness of powerful first-order abstraction levels comes along with increased computational efforts for algorithms working with first-order abstraction levels, especially at PIAGET-3 where logical ML algorithms such as ILP are typically employed. On the other hand, the nature of first-order logical languages introduces many possibilities to use and learn *domain knowledge* in sequential decision making problems, and fortunately many logical ML algorithms naturally incorporate mechanisms to use (declarative) knowledge in the learning process and to transfer learned knowledge to other problems.

All in all, it turns out that in the newly introduced relational context, we can use many established results from Chapters 2 and 3. Still, there are many new opportunities and challenges that arise due to a new representational framework. In the following sections we elaborate on both new representational and new algorithmic aspects. We focus on a number of frequently returning patterns, and refer to the next chapters for details on individual methods.

4.5.2.1 REPRESENTATIONAL ASPECTS

Section 4.2 has treated logical KR formalisms that are of use for abstraction and generalization in relational RL. The majority of approaches employs purely symbolic, logical languages, though some additional kernel-based and sub-symbolic (e.g. neural network based) have recently been developed for first-order domains. There are many trade-offs to be made when working with logical formalisms (see Section 4.2.2), especially for compu-

tationally demanding tasks such as relational RL. Due to recent developments in first-order theorem proving and model-checking, the use of full FOL languages might be considered (see Sanner and Boutilier, 2006, for a good example) as a standard language, in spirit of the definition of the FORM framework. However, most relational RL approaches use restricted formalisms, most often for reasons of computational efficiency.

Logical Structures and Parameters. As RMDPs introduce probabilities and utilities, abstractions levels are made up of logical structures and parameters. Relational RL shares with SRL (De Raedt and Kersting, 2003, 2004, and see Section 4.3.3) the basic distinction between learning (and reasoning with) structured (e.g. logical) components and the optimization of parameters of these structures. However, the context of relational RL introduces additional challenges due to a bias towards *active learning* and *concept drift* (Maloof, 2003). In a basic RL setting there is no such thing as a data set on which a probabilistic logic model is to be fit (as in SRL). Instead, samples are gathered through interaction with the environment based on the current policy. Learning algorithms progress through a series of policies and value functions which might differ considerably each time slice. In contrast to approximation architectures such as MLPs, there are very few algorithms that support incremental learning of logical structures. Either, one fixes a logical structure a priori and focuses on parameter learning (e.g. Morales, 2003) or one may induce new logical structures that replace the current structures after certain interaction intervals (e.g. Džeroski *et al.*, 2001a; Fern *et al.*, 2007). *Incremental* learning of structures is possible by *refining* the current structure (Driessens *et al.*, 2001a, e.g.), or by rearranging the current structure for example by tree restructuring (Dabney and McGovern, 2006, 2007).

Efficient Data Structures. The increased computational complexity of learning and reasoning in first-order domains makes efficiently representing and storing structures such as value functions a pressing issue, more than in propositional contexts. First-order formulas enable compact abstractions over objects, but one needs to *store* these abstractions in a compact way and in Section 4.2.3.1 we have discussed some examples. Many relational RL approaches rely on the availability of efficient data structures, such as ADDs (e.g. Joshi *et al.*, 2006; Wang *et al.*, 2007), graphs (Dabney and McGovern, 2006, 2007) and trees (Džeroski *et al.*, 2001a) to compactly represent essentially sets of first-order formulas.

Compact data structures are necessary, but not enough. In addition, one needs efficient *operations* for storage and retrieval, but also for matching formulas to ground states, for manipulating structures and for reducing the size of data structures. For instance, in the context of clausal logic, several efficient matching (e.g. subsumption) procedures have been developed (Kietz and Lübbe, 1994) and used in relational RL (e.g. Skvortsova, 2006b). Additional efficient inference optimizations that have been used in relational RL include tabling (see Chapter 6) and query packs (e.g. see Driessens *et al.*, 2001a). In the same setting, efficient *reductions* of clauses can be computed to obtain smaller, but logically equivalent, clauses (Gottlob and Fermüller, 1993). The availability of such operations enabled the model-based algorithm REBEL (Kersting *et al.*, 2004) to be practically feasible, whereas a similar algorithm SDP (Boutilier *et al.*, 2001) was not, due to the full FOL language on which it was based where these operations were not readily available. The field of ILP (see Section 4.3.2) has studied many efficiency improvements for working with (clausal) logical representations (e.g. see Struyf, 2004).

Background Knowledge and Domain Theories. Employing unified languages such as FOL enables (declarative) specification of additional knowledge about a problem and the use of that knowledge to speed up learning, or to tackle more complex problems. The role of domain theories or background knowledge is relatively new in RL. Early RL research focused on *tabula rasa* learning (i.e. starting from scratch) and more recent work has focused on finding (propositional) structure in models and algorithms (see Chapter 3). Explicit use of language extensions (e.g. as in background predicates in ILP) or domain theories (e.g. ramifications and constraints) in RL has recently received attention because of the use of more expressive, first-order formalisms.

Most relational RL work has employed background predicates to enrich the representation language for states, transition models, value functions and policies. Many inductive methods are dependent on the availability of such predicates (e.g. Džeroski *et al.*, 2001a; van Otterlo, 2004a) or on more complex language and search biases (e.g. Cole *et al.*, 2003)⁴⁷. Others are less dependent⁴⁸ on such predicates due to a different choice in language and induction technique (e.g. using taxonomic syntax, see Fern *et al.*, 2007). Other exceptions are formed by instance-based (Driessens and Ramon, 2003) and kernel-based (Gärtner *et al.*, 2003) methods that do not rely per se on available background predicates, but these methods do rely on the availability of a (domain-dependent) distance function or kernel definition. Domain theories, constraints and ramifications are vital for model-based approaches such as REBEL (Kersting *et al.*, 2004) and SDP (Boutilier *et al.*, 2001), see Chapter 6. Because most relational RL methods borrow techniques from ILP and action formalisms, results in these fields on how to incorporate knowledge in reasoning and learning, directly carry over to relational RL and extend its potential considerably. On the one hand, this requires additional reasoning components to be incorporated into learning algorithms, but on the other hand it brings the unification of learning and reasoning closer (Dietterich, 2003, and see also Chapter 7).

FOL approaches in RL also introduce a convenient way to specify and use domain (e.g. transition) models, program constraints (e.g. policies, hierarchical task decompositions) and other, more complex, *mental models*. For instance, supplying a hand-coded *guidance policy* can be useful when available (Driessens and Džeroski, 2002b). Domain models provide means for model-based algorithms (see Chapter 6), but also for additional *heuristic search* and *planning* algorithms to *help* the learner (e.g. Gardiol, 2003; Fern *et al.*, 2004a; Karabaev and Skvortsova, 2005). Central issues are how to use and learn as much domain knowledge as possible in order to aid the learner, and incorporate RL into complex agent architectures that can handle complex tasks but lack learning (van Otterlo *et al.*, 2003, 2007, and see more in Chapter 7).

4.5.2.2 ALGORITHMIC ASPECTS

Much of the new algorithmic aspects of relational RL can be understood as *lifted* algorithms developed in the propositional setting. In the first-order setting, various forms of deductive and inductive techniques for logical structures from Sections 4.2 and 4.3 are

⁴⁷Higher-order logic approaches in ML such as this are very much dependent on language and search bias, especially when compared to standard ILP settings.

⁴⁸The price Fern *et al.* (2007) pay for this, is that their language is restricted to unary predicates. And although domain descriptions can be transformed into such representations, this typically blows up the syntactic representation.

combined with parameter learning methods derived from the RL algorithms in Chapter 2, mimicking algorithms from Chapter 3. Important issues are the ratio between deduction (i.e. reasoning) and induction (e.g. learning), and the ratio between sampling RMDPs and logical structures on the one hand, and sampling *within* a single RMDP (e.g. as in classical RL algorithms).

Structures and Parameters. Much of the work in relational RL employs some form of logical structure, augmented with parameters. And because in the relational setting, universal (practical) learning architectures such as MLPs do not exist, structure learning is an integrated part of most algorithms, i.e. the *structural credit assignment problem* (see Section 2.1) has to be solved alongside with the temporal credit assignment problem. An insightful way to see it, is to view the structural learning part of relational RL again as a trade-off, now between keeping the current abstraction level (*exploitation*) and moving to a new, modified abstraction level (*exploration*), see Figure 4.12. The classical exploration–exploitation trade-off deals with the choice between acting according to the current best policy and exploring new actions (see Section 2.1). This new trade-off is defined on the level of abstraction levels (see also the discussion at the end of Chapter 3).

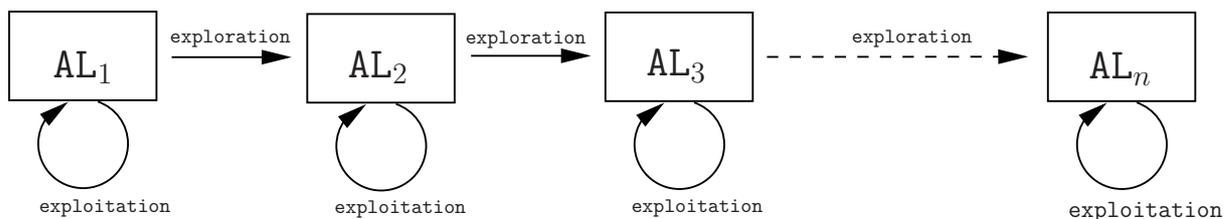


Figure 4.12: The exploration and exploitation of abstraction levels in relational RL.

There are roughly three different approaches that deal with this trade-off, i.e. that interleave learning of both problems in relational RL.

The first approach is by defining all logical structures *a priori* such that learning is focused on the parameters (e.g. PIAGET-1). This avoids complicated structure learning algorithms and relies on the availability of other means (e.g. definition by a designer, or methods that compute the structural from the problem definition), but, in return, provides better means for convergence analysis (e.g. Kersting and De Raedt, 2004) or performance bounds (e.g. Guestrin *et al.*, 2003a; Sanner and Boutilier, 2005). Examples of automated construction include the *a priori* (e.g. PIAGET-0) induction of first-order features (Walker *et al.*, 2004) or general ILP algorithms and behavioral cloning (Morales, 2004a,b).

The second approach consists of *batch-like* (or, *supervised*) approaches, either iterative or single-time. For example, if one has access to a dataset of samples of optimal state–action pairs, an (optimal) abstract policy can be obtained by supervised learning on this set (Kharden, 1999a,b). Several systems obtain such a dataset by applying classical solution techniques (e.g. DP, or planning) in the ground RMDP (e.g. Lecoecuche, 2001; Yoon *et al.*, 2002; Mausam and Weld, 2003; Cocora *et al.*, 2006). In most cases, the dataset is not available from the start, but several subsequent datasets can be acquired by interaction with the RMDP. Learning in this case, proceeds by an interaction that gives rise to a set d_1 of examples, from which a policy π_1 is learned. Following π_1 , a next dataset d_2 is obtained, giving rise to a policy π_2 and so on. Iterative procedures such as this in relational RL

(Džeroski *et al.*, 1998; Fern *et al.*, 2007) show many similarities with two-phase learning methods in SRL, or with structural EM approaches (Friedman, 1998).

A third approach is to *modify* logical structures while learning, instead of constructing them from scratch again and again. Truly incremental logical learning algorithms are not available, though several techniques used in relational RL *extend, refine* (Driessens *et al.*, 2001a; Sanner, 2005, 2006a) or *modify* logical structures on the basis of new examples. The volatile nature of interaction data in RL environments limits the applicability of refinements of fixed logical structures, and for this reason some methods try to completely *restructure* the current structure (Dabney and McGovern, 2006, 2007). Extensions of logical structures can also be obtained by deductive methods such as *regression* (Gretton and Thiébaux, 2004a; Sanner and Boutilier, 2006). Evolutionary learning methods for relational RL can be seen as extending the logical structures online, though they are based on entire *populations* of similar structures (Mellor, 2005a,b; Muller and van Otterlo, 2005).

Learning Control vs. Learning Representations. Relational RL, either model-free or model-based, has to cope with *representation induction* and *control learning*. A starting point is the RMDP with associated value functions V and Q and policy π that correspond to the problem, and the overall goal of learning is an (optimal) abstract policy $\tilde{\Pi}$.

$$\text{RMDP } M = \langle S, A, T, R \rangle \xrightarrow{\text{control learning} + \text{representation learning}} \tilde{\Pi}^* : S \rightarrow A \quad (4.8)$$

Because learning the policy in the ground RMDP is not an option, various routes can be taken in order to find $\tilde{\Pi}^*$. One can first construct *abstract value functions* and deduce a policy from that. One might also inductively learn the policy from optimal ground traces. Relational abstraction over RMDPs induces a number of *structural* learning tasks.

A characteristic pattern in model-based approaches (see Chapter 6) is the following series of purely deductive steps:

$$\begin{array}{ccccccc} \tilde{V}^0 \equiv \mathcal{R} & \xrightarrow{\mathbf{D}} & \tilde{V}^1 & \xrightarrow{\mathbf{D}} & \tilde{V}^2 & \xrightarrow{\mathbf{D}} & \dots & \xrightarrow{\mathbf{D}} & \tilde{V}^k & \xrightarrow{\mathbf{D}} & \tilde{V}^{k+1} & \xrightarrow{\mathbf{D}} & \dots \\ \downarrow \mathbf{D} & & \downarrow \mathbf{D} & & \downarrow \mathbf{D} & & & & \downarrow \mathbf{D} & & \downarrow \mathbf{D} & & \\ \tilde{\Pi}^0 & & \tilde{\Pi}^1 & & \tilde{\Pi}^2 & & & & \tilde{\Pi}^k & & \tilde{\Pi}^{k+1} & & \end{array} \quad (4.9)$$

The initial FORM specification of the problem is taken as a logical theory. The initial reward function \mathcal{R} is used as the initial zero-step value function \tilde{V}^0 . Each subsequent abstract value function \tilde{V}^{k+1} is obtained from \tilde{V}^k by *deduction* (\mathbf{D}). From each value function \tilde{V}^k a policy $\tilde{\Pi}^k$ can be deduced. No sampling is required, and an optimal value function can be derived solely by deduction from the FORM.

A large majority of the work in relational RL samples the underlying RMDP to *estimate* values for the current structures and uses this information for the induction of new structures or the modification of current ones. A typical Q -learning pattern is the following:

$$\begin{array}{ccccccc} \tilde{Q}^0 & \xrightarrow{\mathbf{S}} & \{\langle s, a, q \rangle\} & \xrightarrow{\mathbf{I}} & \tilde{Q}^1 & \xrightarrow{\mathbf{S}} & \{\langle s, a, q \rangle\} & \xrightarrow{\mathbf{I}} & \tilde{Q}^2 & \xrightarrow{\mathbf{S}} & \dots \\ \downarrow \mathbf{D/I} & & \parallel & & \downarrow \mathbf{D/I} & & \parallel & & \downarrow \mathbf{D/I} & & \\ \tilde{\Pi}^0 & \longrightarrow & \{\langle s, a, q \rangle\} & & \tilde{\Pi}^0 & \longrightarrow & \{\langle s, a, q \rangle\} & & \tilde{\Pi}^0 & \longrightarrow & \{\langle s, a, q \rangle\} \end{array} \quad (4.10)$$

An initial abstract Q -function \tilde{Q}^0 (e.g. a first-order tree with only a *root* node) is used to sample (\mathbf{S}) state–action pairs with corresponding Q -values, by interacting with the environment. These samples are then used to induce (\mathbf{I}) a new Q -function \tilde{Q}^1 . A variation on

this scheme is to induce a policy $\tilde{\Pi}^n$ from a Q -function \tilde{Q}^n and derive the samples from the policy. The reason for doing this is that the policy presumably generalizes better over unseen parts of the state space, especially early in the learning process (e.g. see P -learning in Džeroski *et al.*, 2001a).

A third general pattern in relational RL can be characterized as a standard *policy search* (see Section 3.7) process. One starts with a policy structure $\tilde{\Pi}^0$, generates samples by interaction with the underlying RMDP, generates a new abstract policy $\tilde{\Pi}^1$, and so on.

$$\tilde{\Pi}^0 \xrightarrow{\mathbf{S}} \{\langle s, a, q \rangle\} \xrightarrow{\mathbf{I}} \tilde{\Pi}^1 \xrightarrow{\mathbf{S}} \{\langle s, a, q \rangle\} \xrightarrow{\mathbf{I}} \tilde{\Pi}^2 \longrightarrow \dots \quad (4.11)$$

An important difference with Equation 4.10 is that there are no explicit representations of \tilde{Q} involved. It essentially transforms the RL process into a sequence of supervised learning tasks, and has proved very useful in relational RL (Fern *et al.*, 2006). Evolutionary policy search in relational RL follows this scheme, except that instead of one current policy $\tilde{\Pi}^n$, these systems keep a set of policies and policies are usually modified instead of induced from scratch.

One additional possibility that can be used in the general patterns of Equations 4.9 to 4.11 are *ground* policies and value functions. Samples obtained from a solved, ground RMDP can replace the sampling procedures used by other methods. Obviously, this can only be used for very small RMDPs, but an advantage is that one can obtain *exact* value functions and optimal policies using classical solution techniques from Chapter 2. For example, Lecoecue (2001) solves small ground RMDPs and feeds the optimal state-action pairs into a decision rule learner to obtain an optimal policy, while Cocora *et al.* (2006) do the same using decision trees. Mausam and Weld (2003) propose similar techniques for value functions. Similar techniques have been employed using a planner to supply state-action examples in deterministic (Kharon, 1999a,b; Martin and Geffner, 2000, 2004) and stochastic (Yoon *et al.*, 2002) domains.

Regardless of whether the learning process is deductive or inductive, we see that learning generates structures and estimates values for these structures in a continual process (see Equations 4.9– 4.11). Therefore, the *convergence* of learning algorithms can be analyzed on two levels. The first is a classical view on convergence (see Section 3.6.2.4); whether the estimation of values for a fixed structure converges. The second type of convergence is on the *structural level*; will the algorithm ever arrive at a stable final structure (e.g. $\tilde{\Pi}^*$) after several structural induction steps? Or, relating to Figure 4.12, when to stop exploring new abstraction levels? This is an important, but largely unanswered question in relational RL (but see Ramon, 2005b,a; Fern *et al.*, 2006, for initial progress). Furthermore, the richness of FOL allows for a wide variety of semantically equivalent abstract policy representations, such that *compactness* or *comprehensibility* can function as additional criteria. For the model-based setting in Chapter 6 we will see that structural convergence can be characterized somewhat more rigorously.

Efficient Sampling and Approximations. Both the representation space – e.g. of policies and value functions – as well as the RMDP’s state space, are generally extremely large. Efficient techniques are needed to search both spaces in ways we have described in the previous paragraph in a general way. For the induction of logical structures such as policies from state-action samples, as well as for the sampling process itself, efficient optimizations exist. In addition, they are often orthogonal to the patterns in Equations 4.9– 4.11 and

can be developed and used without changing the particular method. Efficient sampling techniques often correspond to *approximations*, i.e. a trade-off between the amount of sampling and performance or accuracy is introduced. We can distinguish between *algorithmic approximations* and *representational approximations*.

- **Algorithmic Approximation: Sampling FOR Structures.** In several steps in Equations 4.9– 4.11 structures are induced from samples. Important questions are about where the samples come from, and whether one could do with less (possibly more accurate) samples. Some work has computed PAC⁴⁹ bounds on the number of samples needed (Khargon, 1999b,a; Guestrin *et al.*, 2003a; Fern *et al.*, 2006) in different settings. However, all induction processes in relational RL need to determine when to move to a new abstraction level, and most often heuristics are used.

Most model-free algorithms assume irreversible world experiences (i.e. standard RL interaction with the environment), whereas Fern *et al.* (2006) assume an unconstrained world simulator that can deliver samples at any requested state. This enables to use *policy rollout* as yet another technique to obtain more accurate samples. Driessens and Džeroski (2004) supply a hand-coded *guidance policy* that biases the sampling, creating possibilities for *shaping* when the amount of guidance is varied. Croonenborghs *et al.* (2007b) use an approximate (learned) model as a look-ahead function to obtain more accurate samples of Q -values. Much of the work on search and exploration in RL (see Section 2.6.3) can be reused for relational RL, e.g. see the work by Gardiol and Kaelbling (2003) on using envelopes and search for RMDPs.

- **Representational Approximation: Sampling OF Structures** Representational approximation can be obtained by deliberately⁵⁰ using approximate logical structures that trade-off their complexity and size with their computational complexity of reasoning and induction. In addition, instead of focusing on sampling in the classical RL sense, i.e. sampling states, actions or Q -values, considerable improvements can be obtained in the *sampling* of logical structures themselves. Approximate representations and structure sampling are important for computationally demanding tasks such as relational RL.

What concerns approximate representations, there are several examples, such as graph kernels and Gaussian processes (Gärtner *et al.*, 2003), naive Bayes representations (Sanner, 2005, 2006a), instance-based representations (Driessens and Ramon, 2003), and first-order features (Walker *et al.*, 2004; Sanner and Boutilier, 2005, 2006). An interesting combination is TRENDI that uses an exact, decision tree but employs an approximate, instance-based generalization in the leaf nodes. Zettle-moyer *et al.* (2005) use an approximate representation of 'noisy' outcomes of actions, such as the result of knocking over a BLOCKS WORLD tower.

As for structure induction, many ILP methods employ biases and heuristics to guide and constrain the induction of structures (see Section 4.3.2). Both Dabney and McGovern (2006, 2007) and Walker *et al.* (2004) use the stochastic sampling techniques

⁴⁹PAC: Probably Approximately Correct (e.g. see Alpaydin, 2004).

⁵⁰Meaning that these methods deliberately use representations that are strictly too simple to capture the full, intended semantics. In essence, most representations do approximate, but because only a limited amount of sampling is used to learn them.

from Srinivasan (1999). The first however, use it for limiting the number of possible new tests considered for the expansion of an abstract Q -function, whereas the second use it for constructing a set of first-order features over the RMDP's state space. Sanner (2006a) uses an APRIORI-style association rule mining technique to focus structure generation on portions of the state space that are frequently visited.

Deductive sampling is used by Gretton and Thiébaux (2004a) to generate logical structures for induction of an abstract value function using the HOL decision tree learner ALKEMY (Lloyd, 2003). A similar technique was used by Sanner and Boutilier (2006) to generate additional features in an API framework. Another useful way to focus the generation of structures is by limiting the state space to be only that part that can be visited using heuristic search (Karabaev and Skvortsova, 2005).

All in all, representational approximations are aimed at representing and generating only those structures that are likely to be important for good performance. There are many other techniques that can be applied in relational RL to improve efficiency.

Sampling RMDPs: Know Thy Population Size. One of the great promises of relational RL is generalization over objects, and with that, transfer of solutions to other, similar problems. As can be seen from Figure 4.11, a FORM generates a family of RMDPs ranging from very small to very large (or even infinite). This introduces the opportunity to choose a 'simple' instance from this family, quickly learn a suitable policy and transfer this policy to larger domains. For instance, a policy $\tilde{\Pi}$ for stacking all blocks into one tower can be learned in a 3-block world, but presumably used in worlds with hundreds of blocks. Indeed, this has been recognized in relational RL and many methods make use of this.

More formally, let \mathcal{F} be a FORM and let \mathcal{M}_a be the family of all RMDP modeled by \mathcal{F} differing only in the domain size. The *speedup ratio* (Walsh *et al.*, 2006) is defined as:

$$\frac{\mathbf{E}_{M_t \sim D}\{T(M_t)\}}{\mathbf{E}_{\mathcal{M} \sim D, M_t \sim D}\{T(M_t|\mathcal{M})\}} \quad (4.12)$$

where $T(M_t)$ is the time needed to find an optimal policy in our target instance M_t , and $T(M_t|\mathcal{M})$ is the time needed to find an optimal policy if information is transferred from \mathcal{M} , containing a set of m instances sampled from \mathcal{M}_a , following distribution D . In other words, a speed-up can be gained by looking at other, similar problems which are – in practice – usually much smaller. Quoting Baum (2004, p. 154): *"To extract structure, one must look at classes of problems"*. Most relational RL methods claim the ability of transfer, but all are based on the natural capabilities of object-based generalization and the fact that the environments considered behave 'similarly' when the number of domain objects is varied. A key assumption is interchangeability between objects of the same class. Some methods use this insight to exactly solve very small ground RMDPs and generalize from that (see previous section).

Policy-based methods are more suitable for transfer because they naturally generalize over instances. Value-based methods are more problematic, because they imply a different solution per member of the family. All methods that estimate abstract Q -value functions from multiple instances simultaneously (e.g. Džeroski *et al.*, 2001a; Mausam and Weld, 2003) are vulnerable to large variations in the estimated values. Guestrin *et al.* (2003a) do provide bounds on policy quality when it is derived from such a Q -function, but these are PAC-bounds under the assumption that the probability of domains falls off exponentially

with their size. In contrast, Sanner and Boutilier (2005) derives strict bounds on the greedy policy derived from an approximated value function. However, this bound may still be the complete reward range in the case of universal quantification in goals. Evolutionary approaches (see Chapter 5) go even one step further than policy-based approaches such as API (Fern *et al.*, 2006). That is, their only performance criterion is the total reward intake, which has – from the learner’s perspective – nothing to do with the domain size. On the other hand, they suffer from large variation in the feedback signal.

Similar problems when working with different or unknown domain sizes arise in the context of *lifted* first-order probabilistic reasoning (de Salvo Braz *et al.*, 2005), where the goal is to do all inference on an abstract logical level instead of doing it eventually on ground level (as is common practice in SRL, see Section 4.3.3). This is related to challenges in model-based relational RL that work on an abstract level too but have difficulties with universal reward functions that depend on the domain size.

Transfer as a separate topic is an active ML research area (see for some further discussion Chapter 7). Although transfer is a natural phenomenon in relational RL, recently some progress was made in explicit studies (Mellor, 2005b; Driessens *et al.*, 2006a; Stracuzzi and Asgharbeygi, 2006; Walsh *et al.*, 2006).

4.5.3 Learning and Representation Tasks in Relational RL

Here we will briefly outline some new aspects when the five abstraction types defined in Chapter 3, are lifted to first-order domains.

4.5.3.1 STATE SPACES

State abstraction (see Section 3.4) forms an integral part of relational RL, because ground RMDP state spaces are usually too large to handle. Yet, in most cases they are not represented explicitly (see also FORMs in Definition 4.5.5), but more implicitly in abstract value functions, policies and transition functions (but see Morales, 2003; van Otterlo, 2004a). Logical abstractions over state spaces can be represented in various forms (see Section 4.2.3). A new aspect in relational RL are large, structured *action spaces*, usually combined with state space abstraction. This is because abstract specifications of actions typically involve specifications that connect action *variables* to objects mentioned in pre-conditions and properties of actions. All in all, state abstraction is present in most methods and in Chapter 5 we describe CARCASS (van Otterlo, 2003, 2004a) and other methods that deal explicitly with state (and action) space abstraction.

4.5.3.2 FACTORED REPRESENTATIONS

Factored representations of MDPs are structured, propositional representations that make use of redundant structure in transition functions, value functions, state spaces and policies. FORMs (see Definition 4.5.5) are a first-order counterpart of this type of representation. They provide the same kind of abstraction over RMDPs. A crucial difference is that FORMs naturally represent families of RMDPs whereas propositional factored representations usually target a specific MDP. Nevertheless, model-based techniques in relational RL typically mimic the propositional solution techniques we have described in Section 3.5, and we will deal extensively with REBEL (Kersting *et al.*, 2004; van Otterlo *et al.*, 2004) and other structured solution algorithms for RMDPs in Chapter 6. These methods rely

heavily on *deductive* techniques, using the FORMs as a kind of logical theory. Factored relational representations typically work at PIAGET-1, where learning value learning takes place on fixed structures (e.g. Guestrin *et al.*, 2003a; Sanner and Boutilier, 2005), or at PIAGET-3, where structure and value learning are taking place simultaneously. PIAGET-0-type learning corresponds to learning transition models over first-order domains (Pasula *et al.*, 2004; Zettlemoyer *et al.*, 2005), and we deal with these methods in Chapter 7.

4.5.3.3 VALUE FUNCTION APPROXIMATION

Value function approximation (VFA) (see Section 3.6) maps states and state–action pairs into real numbers. In the propositional setting a multitude of techniques is available as approximation architecture, most of them relying on the Euclidean structure of the input space (e.g. *fence-and-fill learning*, see Section 3.3.3.2). In the relational setting, which is more symbolic in nature, the input space (e.g. consisting of states and actions) is ordered by entailment and other techniques must be used to learn regression mappings. There are not many robust regression techniques in the first-order setting, especially not when data is constantly changing, as is the case in RL. But, in Chapter 5 we will discuss a number of approaches that have been developed in the context of relational RL. Among these methods are decompositions in terms of first-order features, distance functions between first-order structures, first-order kernels and first-order decision tree algorithms. The algorithmic of VFA in relational domains is very similar to that of the methods described in Section 3.6, once the representational level at which value learning takes place, is defined. Most methods work at PIAGET-1, learning value functions over fixed abstraction levels, or at PIAGET-3, learning both structures and value functions.

A crucial difference with standard, propositional VFA is that at some point, logical structures have to be generated, either before learning as first-order features, or during learning by introducing new structures or features. There are no (practical) general-purpose approximators such as MLPs for relational data. A second difference is the semantics of value functions in the relational contexts, because learning can take place over complete families of RMDPs that all have a different (ground) value function. A final difference with the propositional context is that convergence results, in the context of structure learning as integral part of VFA, are not yet studied (but see Ramon, 2005a; Ramon and Driessens, 2004). Traditional convergence analysis (e.g. see Bertsekas and Tsitsiklis, 1996) is not concerned with the structure learning aspects. Many of the results in propositional state abstraction (e.g. Li *et al.*, 2006) can be used in the relational context, but the additional aspects of action aggregation, families of RMDPs, infinite RMDPs and the use of structural induction algorithms, introduce new challenges that must be solved.

4.5.3.4 POLICY SEARCH

In Section 3.7 we have described some methods that search for policies directly. Because policies are able to generalize across a problem domain, they can be very useful to deal with families of RMDPs. A second advantage of policy search in relational domains is that abstract policies for RMDPs are typically very compact, and usually orders of magnitude smaller than abstract value functions for the same problem. Techniques for policy search in some policy space can relatively simply be lifted to first-order domains. On the other hand, policy search based on policy gradient techniques are less suitable for the relational context because smooth mappings of states to actions (such as done by neural networks)

are not available in the relational context (but see Itoh and Nakamura, 2004; Gretton, 2007a). One direction we will discuss in Chapter 5 is searching for policies based on datasets consisting of state–action samples that may be optimal (e.g. obtained from a supervisor, or from an external planning algorithm) (e.g. Khardon, 1999a,b), or obtained by sampling the current policy (e.g. Fern *et al.*, 2007).

A second direction we will describe in Chapter 5 consists of methods that are based on evolutionary search, which shares with ILP methods that a search in the space of logical policy representations can be performed, but differs from ILP in that it uses a more heuristic way of searching the space and that it is better suited to the limited amount of feedback in an RL environment. There we will introduce GREY, a new evolutionary search method for relational RL, as well as other methods for population-based RL.

4.5.3.5 HIERARCHICAL

Hierarchical approaches to relational RL are not much different from classical hierarchical methods discussed in Section 3.8. Once all representational features are lifted to the first-order case, HRL algorithms can be applied. Although some hierarchical decompositions support simple parameterizations of sub-policies, first-order languages have a more general support for this because they contain variables that can be shared among nodes in a hierarchical task or policy decomposition. Hierarchical approaches are also related to the connection between *learning and reasoning*. Using learned task hierarchies in *plan libraries* with corresponding reasoning mechanisms in logic-based agent architectures can be seen as bridging the gap between learning and reasoning (van Otterlo *et al.*, 2003). *Program constraints* in logical agents can be used to *bias* relational RL, by defining a set of logical behaviors (sub-policies). From all directions in relational RL, hierarchical approaches have been given the least attention so far. In Chapter 7 we will discuss some initial approaches, as well as how (hierarchical) relational RL approaches fit into more complex architectures and agents.

4.5.4 What is Relational RL?: Different Viewpoints

Now that we have showed that relational RL combines many representations and algorithms from various fields, it is time to ask what relational RL actually is. This depends on your viewpoint. There are at least four views on relational RL that matter.

Relational RL is Lifted RL. The view (see also Kaelbling *et al.*, 2001; van Otterlo, 2005) we have propagated throughout this chapter, is that relational RL can best be seen as *lifting* the whole range of RL algorithms found in Chapters 2 and 3 to relational domains (see also van Otterlo, 2008b, for related discussion). To do this, one has to lift the KR framework to the relational case, using first-order logic, logical ML and logic-based action formalisms. This whole enterprise is very much related to the way many propositional ML algorithms have been lifted to first-order domains (see the methodology in van Laer, 2002, and further Section 4.3.2)

For model-free settings, the first approach (Q-RRL) was reported by Džeroski *et al.* (1998) who used a combination of standard *Q*-learning with a logical regression engine algorithm. The start of model-based algorithms was given by Boutilier (2001)'s work on a value iteration algorithm using a SC formalization (SDP). These works can be seen defining landmarks for the model-free and model-based settings (van Otterlo, 2002). In

Chapter 5 we will introduce several new model-free algorithms, and in addition survey a whole range of other methods. In Chapter 6 we introduce the new model-based algorithm REBEL and survey other existing approaches. Interestingly, most model-based algorithms follow closely SDP's outline whereas in the model-free setting the work is much more diverse.

Relational RL extends SRL. From a related point of view, relational RL can be seen as extending the field of SRL methods (see Section 4.3.3). SRL already combines *logic*, *learning* and *probability* (De Raedt and Kersting, 2003, 2004) and relational RL provides in the addition of a *utility framework*. Furthermore, relational RL adds *utility-based* and *active learning* to an existing range of supervised and unsupervised learning methods in SRL. Some relational RL approaches incorporate SRL learning algorithms such as PRMs (Guestrin *et al.*, 2003a), probability trees (Croonenborghs *et al.*, 2007b) and the induction of stochastic transition functions (Pasula *et al.*, 2004; Zettlemoyer *et al.*, 2005). Full integration of utility in SRL would require a complete characterization of how deduction, induction, probability and utility interact on both the semantic and syntactic level, and is still an open problem (but see Sato, 2001; Asgharbeygi *et al.*, 2005, for some examples in deductive algorithms).

Relational RL reunites UAI and ICAPS. Relational RL stands in between learning and planning. The UAI⁵¹ community has developed many ML techniques, firmly rooted in statistics, but is largely limited to at most propositional representations. On the other hand, the ICAPS⁵² planning community has been using first-order specifications for a long time, but mainly focused on deterministic problems, possibly augmented with 'logical' versions of uncertainty such as *disjunctive* uncertainty. Relational RL can be seen as standing in the middle, combining planning approaches with uncertainty and performance measures (see Mausam and Weld, 2003). This also connects to *heuristics* used in planning (e.g. see Yoon *et al.*, 2005, 2006a,b) and a direction known as *learning to plan* – or, *learning-assisted planning* (Zimmerman and Kambhampati, 2003).

Relational RL is a Key Component of Cognitive Agents. A somewhat broader view on relational RL is to see it as way to provide learning mechanisms for *agents* (van Otterlo, 2002; Džeroski, 2002; van Otterlo *et al.*, 2003; Tuyls *et al.*, 2005; van Otterlo *et al.*, 2007). We can view relational RL as being situated in the context of more general abstraction levels, such as *hierarchies*, *agents* and *multi-agent systems*. One could view relational RL as a sub-component of more complex, intelligent entities (i.e. agents), in which it functions as a method for learning *behaviors*. The agent literature (Wooldridge, 2002; Weiss, 1999; Ferber, 1999) contains many examples of logical agents capable of reasoning, communicating, planning and acting. Relational RL methods can provide a general framework for *learning* in general agent architectures and we describe this direction in Chapter 7.

4.6. Conclusions

This chapter has shown that the complete RL framework defined in Chapters 2 and 3, can be upgraded to solve sequential decision making problems in large, stochastic, first-

⁵¹Uncertainty in Artificial Intelligence community and conference.

⁵²International Conference on Automated Planning Systems community and conference.

order domains. Depending on your view, relational RL can be framed in such diverse areas as planning, statistical relational learning, traditional RL or agent-based approaches (see Section 4.5.4). An interesting question one might ask is whether relational RL is significantly *new* as a research field. Again, the answer depends on your viewpoint, and quite opposite opinions are equally valid.

PROPOSITION 4.6.1 ► Relational RL is either a utility-based extension of probabilistic planning, a utility-based, active learning approach in RL, a new representation scheme in RL or a way to learn behaviors for logic-based agents; as such, it is yet another interdisciplinary direction in AI.

This is indeed a valid point. In fact, this whole chapter has tried to show that when combining methods from logic, ML and planning, a framework for relational RL can be compiled. This view creates many possibilities to learn from the various subfields of AI on which relational RL is founded. An opposing view is the following:

PROPOSITION 4.6.2 ► Relational RL is a new paradigm that aims at solving problems that lie at the heart of AI, namely efficiently learning effective behaviors in the context of logical knowledge representation, uncertainty and limited, evaluative feedback.

Again, this is a valid viewpoint; the combination of all techniques we have surveyed in this chapter, transcends all component parts. It requires that a number of difficult conceptual *and* technical matters are solved in a single framework. We largely agree with the second proposition, but we choose to describe this view from the viewpoint of traditional RL, as we have done throughout Chapters 2 to 4, because it offers a rigorous formal framework to put matters into context (see also recent descriptions in line with this view by van Otterlo and Kersting, 2004; Tadepalli *et al.*, 2004; van Otterlo, 2005).

Relational RL offers many new things compared to the classical RL framework. Among others, we have encountered the following extensions in this chapter:

- A new representational framework that includes all preceding frameworks such as atomic, attribute-value, deictic and propositional representations.
- The ability to use *objects* as first-class citizens in representations of, and predictions about the world.
- The ability to *generalize* over objects and relations in value functions, policies and belief states about the world.
- Parameterizations of policies and value functions, as well as sub-behaviors in behavioral hierarchies.
- The ability to deal with indefinite or even infinite world sizes and learning in worlds without a necessary naming of all objects.
- Excellent opportunities for modular systems of behaviors, or skills, and the possibility of transfer of learned behaviors to other, similar domains or other domains that differ in the number of domain objects.

- Considerable opportunities to use a priori knowledge in the learning process, due to expressive domain modeling languages that can as well be used for giving bias to policies and heuristic search algorithms.

In addition to these advantages, a number of frequently occurring, technical challenges arise that require additional efforts, and for which one can use various methods from fields we have described in this chapter such as planning, KR and logical ML:

- Increased KR efforts required by the more expressive capabilities of FOL.
- Efficient data structures and inference algorithms that lower computational complexity in the average case.
- Similarity measures and distances for first-order structures, such that additional possibilities arise for lifting propositional RL algorithms to the first-order case.
- Investigations into combinations of, and trade-offs between inference and planning on the one hand, and induction and sampling techniques on the other.
- Investigations into SRL techniques that can be used in stochastic domains with concept drift.
- Formal investigations of unified frameworks for the combination of logic, uncertainty, utility and learning, both syntactic and semantic.

In the following three chapters, we describe how these advantages and challenges have been implemented in concrete relational RL systems, roughly along the lines of the five types of abstraction described in Section 4.5.3. This results in the following outline:

- **Chapter 5: Model-Free Algorithms for Relational MDPs**
Model-free relational RL are largely based on inductive techniques, in the absence of a domain model. We formalize the general setting described in Section 4.5.2.2 and propose two new methods: CARCASS, a new model-free algorithm that incorporates Q -learning and model-learning based on fixed abstractions, and GREY, an evolutionary policy search method that incorporates structure learning. Furthermore, we provide an exhaustive survey of other methods.
- **Chapter 6: Model-Based Algorithms for Relational MDPs**
Model-based relational RL can be characterized as *lifted* DP algorithms, that – in principle – can solve FORMs on an entirely abstract level without interaction with the underlying RMDPs. The main modus operandus is deduction. We describe the first implemented version of first-order value iteration, REBEL and provide formal connections with *set-based* DP and *regression planning*. Furthermore, we provide an exhaustive survey of other methods.
- **Chapter 7: Sapience, Models and Hierarchy**
The expressive KR in relational RL offers many opportunities to learn behaviors in behavioral hierarchies and agent architectures. In this chapter we provide this setting and outline initial ideas about the incorporation of relational RL in logic-based agent architectures. Furthermore, we describe connections with other agent languages.

Model-Free Algorithms for Relational MDPs

The core dichotomy in solution techniques for MDPs consists of so-called model-free and model-based methods. This is no different in the relational setting as described in the previous chapter. The current chapter will focus on the model-free methods for relational MDPs. In the first half we focus on value-based methods. We present a new representation, the so-called CARCASS abstraction, that provides a stable, structural representation for RL in RMDPs. We describe an implementation using Q-learning, and additionally we describe how approximate models can be used to make learning more efficient. Experiments and analysis show that learning can be efficient, even for very large environments and that CARCASSs can make elegant use of available domain knowledge. In this part we also provide a detailed survey of other model-free, value-based methods for RMDPs. In the second half of the chapter we focus on methods that side-step value function learning and search in policy space directly. We present first experiences with a novel evolutionary method called EARL GREY, and we survey all other model-free, policy-based methods.

THE MODEL-FREE SETTING in learning sequential decision making problems is commonly referred to as *reinforcement learning* (RL). In this chapter we study this class of algorithms when transferred to first-order domains (see also Kaelbling *et al.*, 2001; van Otterlo, 2002; Džeroski, 2002). The basic setting is one in which an agent *interacts* with an *environment* that is modeled as an MDP, exactly in the way we have described in Chapter 2. The main difference lies in the new type of environments (i.e. RMDPs), now consisting of *objects* and *relations*, and in new types of *actions*, which are now *parameterized* with objects. Without a model, the agent must *explore* these worlds – by trying out actions on objects – and find useful *structural patterns of objects and relations* which can be used as predictors for obtaining *rewards*. For example, in a standard BLOCKS WORLD, the agent might find out that when performing move actions on blocks, the structural arrangement of blocks is altered and that these arrangements (e.g. stacks) are useful predictors for reward. Note that on the one hand, first-order worlds quickly yield huge state spaces, such that the agent must explore many areas of these spaces. On the other hand, by abstracting over objects and relations using logical languages with variables, many new types of structure can be exploited during learning.

The general strategy for RL in first-order worlds is to apply standard RL *algorithms* (as in Chapter 2), and to augment these with *abstraction mechanisms* (as in Chapter 3) that

are *upgraded* to the first-order case. Abstraction is required as a basic component in any relational RL algorithm because of the enormous state spaces. As an example, consider a BLOCKS WORLD RMDP. Standard Q -learning (Section 2.6) can – in principle – be used on the ground RMDP to compute ground value functions and policies. But, taking inspiration from propositional-based abstraction using *trees*, one can approximate a value function using a decision tree that represents the value compactly, for example using the G -algorithm or UTREE (see Section 3.6.2.3). Now, considering the fact that RMDPs use Herbrand interpretations as states, upgrading such a tree-based approach would require a decision tree algorithm that can induce a *first-order logical* decision tree from these interpretations. In Section 4.2.3 we have seen such devices, and one of these is the tree-learner TILDE. Now, when we combine TILDE with Q -learning, we arrive at the very first model-free algorithm for RMDPs, the Q-RRL algorithm (Džeroski *et al.*, 1998).

In fact, most (if not all) model-free relational RL approaches can be seen as first-order upgrades of the algorithms in Chapter 3. Here we describe these approaches (see Tadepalli *et al.*, 2004; van Otterlo, 2005, for surveys) and show that they upgrade the *representational* parts, as well as the *structural induction* parts of standard propositional-based RL algorithms with languages and algorithms derived from the fields of *inductive logic programming* (ILP) (see Section 4.3.2) and *statistical relational learning* (SRL) (see Section 4.3.3). What we also find, is that by replacing these parts with their first-order logical counterparts, many new challenges arise of which some are already present in ILP and SRL. Among these are difficulties with *incremental* and *online* learning, the difficulties of learning both structure and parameters, and the aspects of varying domain sizes.

Goals and Outline of this Chapter. The goals of this chapter are threefold. First, it describes in detail the problems and challenges of solving sequential decision making problems in relational domains (i.e. RMDPs) *in the absence of a domain model*. In Section 5.1 we introduce the setting and highlight some properties, where we emphasize the aspects of *sampling* and *structural induction*. Furthermore, we distinguish between methods that focus on value functions and methods that directly learn policies. We introduce our CAR-CASS algorithm in Section 5.2, which is a model-free algorithm for RMDPs. We describe two variants: one in which a value function is learned using Q -learning, and one in which an approximate model is learned and used to speed-up learning. Both variants work on a priori defined structures, and we analyze some of the properties of the abstractions at the end of the section. In Section 5.3 we relate to other modeling approaches and survey the complete field of model-free, value function approximation methods for RMDPs. The third topic of this chapter is a description of a novel *policy search* method, named EARL GREY, which is discussed in Section 5.4. This algorithm side-steps difficulties with value functions and searches directly in policy space using an evolutionary algorithm. In Section 5.5 we survey all other methods that focus on policies, distinguishing between methods that reduce RL to *classification*, and methods that either use evolutionary algorithms or policy gradient approaches. Finally, we conclude in Section 5.6.

5.1. Model-Free Relational Reinforcement Learning

Although – in principle – first-order domains can be modeled using arbitrarily complex first-order structures, we limit our discussion to fully-observable RMDPs (see Definition 4.1.1). The learning setting for model-free relational RL is the same as for standard

MDPs with the difference now being that the environment is modeled as an RMDP. Algorithm 3 in Section 2.6 contains a generic description of most model-free RL algorithms. In each episode the agent performs an action a in some state s , observes the next state s' and *updates* its internal first-order datastructures, which represent the current value functions Q or V , a model of T and R , or the policy π .

A first difference with the propositional or atomic settings is that each state is now a first-order structure, i.e. a Herbrand interpretation in this case. A second difference is that each action is a ground action atom, containing parameters that are shared by the current state s . The crucial step is the update step, in which the sampled states are generalized into abstract representations of value functions, models and policies. This generalization is typically performed using various ILP/SRL algorithms (see Chapter 4). In model-free relational RL, we distinguish between the following three types of solution techniques.

Value-Based Methods on Static Relational Abstractions. A first class of algorithms consists of models that provide all logical abstractions *a priori*, such that learning amounts to estimating parameters (PIAGET-1) for these logical structures. In Section 5.2 we describe our CARCASS abstraction that supports both direct and indirect, model-free RL over a given relational abstraction over the joint state-action space. Such algorithms provide a stable learning setting that is comparable to standard (propositional) state aggregations (e.g. see Section 3.4), and by that they enable to carry over convergence results from propositional algorithms. A crucial advantage is that the difficult task of structural induction can be omitted. A downside is that the abstraction levels must be provided before learning, which limits their scope. In Section 5.3 we survey these approaches.

Value-Based Methods with Dynamic Generalization. A more general class of algorithms is formed by those that can dynamically generalize experience (i.e. PIAGET-3) online. These algorithms are in many cases first-order upgrades of the propositional algorithms described in Section 3.6. Methods such as these employ various adaptations of ILP algorithms to induce logical abstractions of value functions in the update step. These methods are fully capable of learning both the representation and the behavior from scratch. Some approaches use first-order *regression* algorithms to learn logical abstractions of value functions. Others have used memory-based approaches or distances and kernels as an alternative to generalization. All such value function approximation algorithms are surveyed in Section 5.3.

Policy-Based Methods. A third class of models is concerned with policies, rather than value functions. As was discussed in Section 4.5.1.2, value functions have limited generalization capabilities when it comes to families of RMDPs. In contrast, policies are often more compact and scale to larger problem instances. Several methods focus on the induction of policies online, mimicking the algorithms in Section 3.7. One type of algorithms uses state-action examples obtained by sampling the current policy as input to ILP algorithms that induce a new, improved policy representation, thereby reducing the RL to a sequence of *classification* problems. Another class of algorithms searches directly in *policy* space, based on the results of *holistic* policy evaluations rather than from individual action effects. In Section 5.4 we introduce our EARL GREY algorithm that uses an evolutionary algorithm to search for good

policies. One additional class upgrades *policy gradient* approaches to the first-order case, and we will survey all policy-based approaches in Section 5.5.

In addition to these three types, an alternative is to first learn a world model, and then use that either in a model-based RL algorithm (see for example in CARCASS in the next section) or in model-based *dynamic programming* algorithms (which are described in the next chapter). Model-learning approaches in first-order domains will be covered in Chapter 7.

5.1.1 Sampling and Structural Induction

One of the fundamental trade-offs in RL is that between *computational complexity* and *sample complexity*. Although only some approaches have studied the latter in the relational setting (e.g. see Fern *et al.*, 2006), we are mostly concerned with the former. Structural induction in first-order domains is a computationally expensive problem. In addition, when employed for online generalization in RL tasks, first-order induction must be capable of dealing with *concept drift* (Maloof, 2003; Widmer and Kubat, 1996), i.e. the fact that the value function or policy *structure* may change over time. A learning process will usually start with an initial policy π (or value function V) that is presumably *structurally* much different from the final, optimal policy π^* (or value function V^*). Samples that are obtained in early stages of learning will not be valid anymore later on, for example because values change or some parts of the state space have been deemed irrelevant for the current task.

The general relational RL setting involves three aspects. First, structural representations of value functions and policies must be induced through interaction with the environment, as in ILP. Second, parameters for these structures, such as probabilities and values, have to be learned in a setting similar to SRL. But, the main characteristic of relational RL is most certainly the fact that the dataset appears to be *non-stable*. As a consequence, either a *series* of batch-like inductive steps is needed on subsequent sets of samples obtained during various learning stages, or new logical structures and accompanying inductive processes are needed that support online, *incremental* learning. Incremental learning amounts to *revision* of logical structures (Greiner, 1999), i.e. changing parts of a structure on the basis of new samples instead of repeatedly inducing a complete structure. In contrast to the propositional setting, very few truly incremental learning algorithms exist for first-order data in the context of uncertainty and utility. The symbolic form of the generalization space in the first-order setting makes it more difficult to compute smooth, incremental changes when generalizing, unlike in the propositional setting where e.g. neural networks can do this quite effectively (see Section 3.6.2.2).

Concerning the sampling process, one might ask where the samples come from, how (and if) they must be stored, and furthermore, whether they can be treated in an incremental fashion or that batches of examples are used at once. In most cases, state-action-value samples are derived from the current Q -value function while exploring the environment using an ϵ -greedy policy. Other options are *policy rollout*, or in case of policy induction, using fully solved ground RMDP instances. A new type of sampling in the first-order setting is *domain instance sampling*, i.e. sampling RMDPs based on varying the domain (size) (Guestrin *et al.*, 2003a), or by generating *random walks* to obtain increasingly more difficult domain instances (Fern *et al.*, 2004a).

Much regardless of how samples are obtained, their actual employment in the induction of logical abstractions can vary among algorithms. The simplest case is when upgrading from ground problem instances (e.g. see also Section 6.5.2.3). All samples are

derived from a (optimally) solved problem and the induction of logical abstractions is only performed once, in a batch fashion. Most other methods use samples in an incremental way. Methods that generalize in the ground space such as instance-based methods like RIB (Driessens and Ramon, 2003), do this by inserting and discarding samples along the way. Roughly, one can distinguish between four types of incremental structural induction and sampling used in relational RL. The first type does not generalize with logical structures containing variables, but – in essence – stays on the ground level. Examples are those based on *distances* between interpretations or *kernels*. Related approaches that *propositionalize* the problem (and perform generalization in a propositional space) have been described in Section 4.1.3.3. All these approaches resort to another generalization space than that of purely logical abstractions. The other three approaches are based on logical induction, and have been described in Section 4.5.2.2. These are based on **i)** fixed logical abstractions, **ii)** *batch*-like dynamic abstractions, and **iii)** truly incremental dynamic abstractions. Differences lie in the sampling process. Whereas in type **ii)**, samples must be gathered and stored, samples in type **iii)** can be discarded immediately. A fundamental difference between the first and the rest of the approaches is that the latter generally deliver *comprehensible* logical abstractions in some language, whereas the first does not.

5.1.2 Representations, and Value Functions vs. Policies

In the model-free setting, a complete model (or axiomatization) of the environment (e.g. as in Section 4.4), neither a FORM (see Definition 4.5.5) is assumed nor required. The only assumption is that there is a simulator that lets the agent interact with the domain via actions, perceptions (i.e. states) and rewards. Still, the agent does need representational structures such as value functions or policies (see Section 4.5.1.2 for general definitions). The specifics of states and actions drives the language for such structures, as well as the induction procedures employed to learn these structures from data. Paying attention to representational aspects is much more important in the first-order setting than in the propositional setting. For instance, when generalizing in the first-order setting, usually one has to specify some kind of *language*, a *language bias* and a *search bias* in order to find good abstraction levels, i.e. to generalize. And even though one does not have to specify full symbolic models (as in the next chapter), such biases will involve many decisions on the language and how it is used. Some approaches side-step these issues by not generalizing in a logical sense, but in return they have to provide (domain-dependent) first-order *distance metrics* or *kernels* which usually involve equally challenging modeling tasks. As in Chapter 2 we can distinguish between several PIAGET-levels: some approaches fix the abstraction level before learning, while others adapt their representation online. In the first-order setting, the latter is more challenging because it involves the induction of logical abstractions from traces generated while learning.

Most of the classical model-free relational RL algorithms focus on learning approximations of value functions. The general structure mimics that of propositional-based algorithms in Section 3.6. However, in the first-order setting, policies are often more desirable because they usually scale easier with domain size, by generalizing over specific objects. A difficulty with policy-based approaches, however, is that they are either based on gradient approaches or on searches in policy space (see Section 3.7). In the context of logical abstractions, gradients (but also distances) are not as natural as in propositional representations, because of the intrinsically symbolic nature of the states and actions. A

general question in the model-free setting is about *what* to represent. Most algorithms explicitly represent value functions and use these to derive policy decisions. In order to get an *explicit* representation of the policy (derived from a learned value function) usually requires additional deductive (or inductive) efforts. For example, a policy can be induced from state-action samples generated using the current value function, or it can be deduced from the value function structure. Doing this *while* learning sometimes pays off, because a policy structure usually generalizes much better than a value function (e.g. see *P*-learning in Džeroski *et al.*, 2001a), thereby biasing future learning experiences. In general, representing either the value function or the policy in ground form is not an option.

5.2. CARCASS: A Model-Free, Value-Based Approach

In this section we introduce the CARCASS¹ abstraction for RMDPs, which is an acronym for *Compact Abstraction using Relational Conjunctions for Aggregation of State-action Spaces* (van Otterlo, 2003, 2004a). According to the descriptions on the previous pages, CARCASS can be classified as a model-free algorithm over fixed relational abstractions. A CARCASS employs an expressive formalism comparable to PROLOG to capture abstractions over states and actions in an RMDP, which can use background knowledge predicates if such domain knowledge is available.

The motivation for this approach is to have a general way of abstraction for use in the solution of RMDPs. Typically, we want to be able to capture state and action aggregation using a logical language, and use these aggregations in model-free algorithms such as *Q*-learning. The CARCASS is general enough to model policies, value functions and even abstract state-action spaces. Learning is focused on estimating values, because the problem of model selection, i.e. generating the abstraction layer itself, is assumed to be solved by other means, for example by a pre-learning phase (e.g. PIAGET-0). The advantages of fixed abstraction layers are that the learning problem is more restricted and stable, and that convergence guarantees are stronger.

Simultaneously and independently, two other approaches were introduced that have many similarities with CARCASS. These are the *relational Q-learning* (RQ) approach by Morales (2003) and the *logical MDP* (LOMDP) framework by Kersting and De Raedt (2003). Much of what we present in this section equally applies to all three methods, and in Section 5.3.1 we will describe some relations and differences between the systems.

5.2.1 Relational Abstractions over RMDPs

A CARCASS employs a logical language to express properties of states and actions. In the following, we assume a general first-order language that includes predicates, constants and variables. Predicates may be basic predicates, for example those that are used in an RMDP definition, or *extended* (or, *background*) predicates. A CARCASS is a form of *abstraction* in which the generalization over states and actions is built-in by a designer. We start by defining the syntax of the CARCASS, after which we provide a semantics based on decision lists. We then show how the abstractions can be used in representing value functions, policies and abstract RMDPs.

¹One of the meanings of the word CARCASS is "the structural framework of a building, ship, or piece of furniture". The intended meaning of the CARCASS abstraction in this chapter is to provide a *structural framework* (a *skeleton* so to speak) of an RMDP, a relational value function or a policy.

Representation (Syntax). The general form of a CARCASS is defined as follows.

DEFINITION 5.2.1 ▶ Let \mathcal{P} be a set of predicate definitions (both basic and extended), \mathcal{D} a set of domain objects and \mathcal{V} a set of variables. A CARCASS \mathcal{C} is a finite, ordered list

$$\left[\langle \mathcal{S}_1, \langle \mathcal{A}_{11}, \dots, \mathcal{A}_{1m_1} \rangle \rangle, \dots, \langle \mathcal{S}_n, \langle \mathcal{A}_{n1}, \dots, \mathcal{A}_{nm_n} \rangle \rangle \right]$$

where each \mathcal{S}_i ($i = 1, \dots, n$) is a conjunction over \mathcal{P} , \mathcal{D} and \mathcal{V} , and all $\mathcal{A}_{i1}, \dots, \mathcal{A}_{im_i}$ are range-restricted action atoms, i.e. $\text{var}(\mathcal{A}_{ij}) \subseteq \text{var}(\mathcal{S}_i)$ for $j = 1, \dots, m_i$ ($i = 1, \dots, n$).

The size of the abstract action space for an abstract state \mathcal{S}_i , denoted $\#A_{\mathcal{C}}(\mathcal{S}_i)$, is the number of actions m_i , and $\{\mathcal{A}_{i1}, \dots, \mathcal{A}_{im_i}\}$ are called the *applicable abstract actions* in state \mathcal{S}_i , denoted $A_{\mathcal{C}}(\mathcal{S}_i)$.

The intuitive meaning of an abstract state in a CARCASS is that it expresses a number of properties of states, with the purpose of aggregating groups of states that are – in some way – similar. Atoms in the states can be negated, and it is assumed that all variables are bound. Each state-actions pair in a CARCASS representation is called a *rule* and can be read alternatively as a Horn clause with multiple heads (a general Horn clause)

$$\mathcal{A}_{i1}, \dots, \mathcal{A}_{im_i} \leftarrow \mathcal{S}_i$$

Each rule can intuitively be read as *if the current state satisfies \mathcal{S}_i (resulting in some substitution θ) then one can apply any of the actions $\mathcal{A}_{ij}\theta$ ($j = 1, \dots, \#A_{\mathcal{C}}(\mathcal{S}_i)$). Ensuring² that the resulting ground actions can, in fact, be applied is left to the designer.*

EXAMPLE 5.2.1 ▶ An example CARCASS abstraction for a BLOCKS WORLD containing three blocks is the following³:

state (\mathcal{S}_1):	$(\text{on}(A, B), \text{on}(B, \text{floor}), \text{on}(C, \text{floor}), A \neq B, B \neq C)$
actions ($\mathcal{A}_{11}, \dots, \mathcal{A}_{13}$):	$\text{move}(A, C), \text{move}(C, A), \text{move}(A, \text{floor})$
state (\mathcal{S}_2):	$(\text{on}(A, \text{floor}), \text{on}(B, \text{floor}), \text{on}(C, \text{floor}),$ $A \neq B, B \neq C, A \neq C)$
actions ($\mathcal{A}_{21}, \dots, \mathcal{A}_{26}$):	$\text{move}(A, B), \text{move}(B, A), \text{move}(A, C),$ $\text{move}(C, A), \text{move}(B, C), \text{move}(C, B)$
state (\mathcal{S}_3):	$(\text{on}(A, B), \text{on}(B, C), \text{on}(C, \text{floor}), A \neq B, B \neq C, C \neq \text{floor})$
actions ($\mathcal{A}_{31}, \dots, \mathcal{A}_{31}$):	$\text{move}(A, \text{floor})$

The first state abstracts over all states in which only two blocks are stacked. In this state, one can put either the block denoted A on C or vice versa. Note that we implicitly assume that there are only three blocks. If there would be more, there could be a block on top of A (or on C) because we have not explicitly specified that it is clear. Also note that in the second state, we could easily replace the six abstract actions by only one, $\text{move}(A, B)$. This is due to symmetrical substitutions; in all states the six actions will induce the same ground action sets. This is an important aspect of relational abstractions, and we will address this later in this section.

²An alternative is to allow illegal actions to be generated, if the simulator can handle these by e.g. not changing the current state and punishing the action with a negative reward.

³Note that we use $A \neq B$ for $\text{not}(A == B)$.

Here we see that abstract states have two purposes. One is that they express certain properties of states, but the other is that they are used to find substitutions for the action variables. By supplying background knowledge definitions, the state abstractions can be very powerful. For example, it is possible to define a predicate `towerHeight(A, H)` relating the top of a tower, A , to the tower height H (see Section 4.2.2.2 for similar definitions).

Here we employ simple conjunctions as a state language, and use PROLOG as a theorem prover, possibly employing background knowledge predicates in the abstractions. However, the general form of a CARCASS is relatively independent of the logical formalism that is used. As long as one can express the properties of the states using a logical formula, and can compute suitable substitutions for the action variables, a CARCASS can be expressed in any language, including rich logics such as the situation calculus (see Chapter 4). Indeed, many other formalisms that have been used for RMDPs are similar to CARCASSs, in that they also provide abstractions over states and actions. For example, a CARCASS definition in terms of first-order trees (instead of decision lists) would be possible too. The specific form that is used here, however, has some computational and modeling advantages that will be made clear in the following sections. Additionally, the type of expressions, and the use of background knowledge predicates, is similar to the ILP setting and this makes the CARCASS representation more amenable to extensions with adaptive abstractions.

Semantics. CARCASSs are not declarative, but an order is imposed on the rules in the CARCASS, rendering it a *decision list* (see Section 4.2.3). The semantics of a CARCASS are given relative to an RMDP or a family of RMDPs (see Chapter 4). Here we focus on one specific RMDP. Let $M = \langle S, A, T, R \rangle$ be an RMDP. Let us first state a standard definition of the semantics of a query based on SLDNF. Let $\$$ be an abstract state. The set of all states covered by $\$,$ denoted $\llbracket \$ \rrbracket$, is the set of states $\{s \mid s \vdash_{\text{SLDNF}} \$\}$. Similarly, one can define partitions over the joint state-action space $S \times A$. The set of all state-action pairs covered by $\langle \$, A \rangle$, denoted $\llbracket \$, A \rrbracket$, is the set $\{\langle s, a \rangle \mid s \vdash_{\text{SLDNF}} \$\theta \text{ and } a \equiv A\theta\}$. The semantics of the rules in a CARCASS is changed by the order that is imposed on the rules.

DEFINITION 5.2.2 ▶ Let $\mathcal{C} = [\langle \$_1, \langle A_{11}, \dots, A_{1m_1} \rangle \rangle, \dots, \langle \$_n, \langle A_{n1}, \dots, A_{nm_n} \rangle \rangle]$ be a CARCASS and let $M = \langle S, A, T, R \rangle$ be an RMDP. The sequential ordering of the rules determines their semantics, denoted $\llbracket \cdot \rrbracket_{\mathcal{C}}$, as follows. For each state $\$_i$ in \mathcal{C} it holds that

$$\llbracket \$_i \rrbracket_{\mathcal{C}} = \llbracket \$_i \rrbracket \setminus \bigcup_{j=1, \dots, i-1} \llbracket \$_j \rrbracket$$

For each state-action pair $\langle \$_i, A_{ik} \rangle$ in \mathcal{C} it holds that

$$\llbracket \$_i, A_{ik} \rrbracket_{\mathcal{C}} = \llbracket \$_i, A_{ik} \rrbracket \setminus \left\{ \langle s, a \rangle \in \llbracket \$_i, A_{ik} \rrbracket \mid s \in \bigcup_{j=1, \dots, i-1} \llbracket \$_j \rrbracket \right\}$$

The set of ground actions covered by an abstract state-action pair $\langle \$_i, A_{ik} \rangle$ in \mathcal{C} when evaluated in a specific state s , denoted $\llbracket \$_i, A_{ik} \rrbracket_{\mathcal{C}}^s$, is defined as $\{a \mid \langle s, a \rangle \in \llbracket \$_i, A_{ik} \rrbracket_{\mathcal{C}}\}$.

Thus, each abstract state in a CARCASS can only cover states that are not covered by states higher in the list, effectively implementing a decision list. Note that in Example 5.2.1 some of the state atoms are unnecessary, either because a general domain theory would make

them superfluous (e.g. that $C \neq \text{floor}$) or because the rule order can be used to make expressions simpler. For example, if state $\$2$ would be placed last, it would be sufficient to represent just two blocks A and B with one action $\text{move}(A, B)$ because all other (three-block) states in which some blocks are stacked, are covered by the other two rules that precede $\$2$. Other, more declarative, interpretations of such lists, where *multiple* rules can cover some specific state, have been used in for example GREY (see later in this chapter), REBEL (see next chapter), and FOX-CS (Mellor, 2007). CARCASSs are targeted at providing abstractions for RMDPs and they simultaneously define state partitions and state-action partitions. Note that for a CARCASS to define a proper partition, usually a *default rule* should be provided at the bottom of the list as a kind of *catch-all* rule to ensure that all state-action pairs in the RMDP are covered. Every RMDP itself can be modeled as a CARCASS in which all states and actions are ground.

Although we have defined CARCASSs as decision lists, depending on the language that is used, one can generate *explicit* partitions by making sure that all abstract states have non-overlapping model sets. In this case, each RMDP state would be covered by only exactly one abstract state. The expressivity of PROLOG supports such definitions of abstract states through background predicates, though in most cases we rely on the order of the list to make expressions simpler. Implementing a CARCASS using a more general logic such as situation calculus would enable explicit partitions more elegantly. Nevertheless, no matter which type of logic is used, the actual modeling of the CARCASS must ensure that the *right* abstraction level is specified. In other words, similar to the standard PROLOG setting (see Section 4.2.2.2), one must ensure that a CARCASS abstracts over the *intended* RMDP (or family). In the following paragraph we describe basic properties of the abstraction in terms of the Markov property, but ensuring that a CARCASS abstracts over the right states and actions for the RMDP at hand, is a delicate task left to the designer.

Abstract RMDPs. Now that we have defined that, and how, CARCASSs induce partitions over both the state space and the state-action space, one can think of at least three applications. Two of these are abstract value functions and abstract policies, and we will describe them in the context of model-free solution algorithms below. The third application is to use a CARCASS to induce an *abstract state-action space*, in a similar way as we have described for propositional state aggregations in Section 3.4. There is, however, a crucial difference with the propositional setting, and that is that in the relational setting *action aggregation* is an explicit aspect of the model.

DEFINITION 5.2.3 ▶ For are given RMDP $M = \langle S, A, T, R \rangle$ (or a family of RMDPs), a CARCASS $\mathcal{C} = [\langle \$1, \langle A_{11}, \dots, A_{1m_1} \rangle \rangle, \dots, \langle \$n, \langle A_{n1}, \dots, A_{nm_n} \rangle \rangle]$ induces a new *abstract* RMDP $M_{\mathcal{C}} = \langle S_{\mathcal{C}}, A_{\mathcal{C}}, T_{\mathcal{C}}, R_{\mathcal{C}} \rangle$. The new state space $S_{\mathcal{C}}$ consists of the state *aggregations* $[\$1]_{\mathcal{C}}, \dots, [\$n]_{\mathcal{C}}$ and the action set $A_{\mathcal{C}}$ is the full ground action set of the original RMDP, but non-zero transition probabilities are only defined for actions in $[\$i, A_{ik}]_{\mathcal{C}}$ for each abstract state-action pair $\langle \$i, A_{ik} \rangle$.

Thus, the applicable ground actions are dependent on the exact state and $T_{\mathcal{C}}$ and $R_{\mathcal{C}}$ are defined in the following way. When aggregating states and actions, the transition probabilities between abstract states are *averages* of the ground transitions in the underlying RMDP. Let $\$1$ and $\$2$ be abstract states and let $A \in A(\$1)$. Now the probability for making a transition from state $\$1$ to state $\$2$ after performing action A is the sum of all transition

probabilities of the ground transitions $T(s, a, s')$ where $\langle s, a \rangle \in \llbracket \mathcal{S}, \mathcal{A} \rrbracket_{\mathcal{C}}$ and $s' \in \llbracket \mathcal{S}_2 \rrbracket_{\mathcal{C}}$, i.e.:

$$T_{\mathcal{C}}(\mathcal{S}_1, \mathcal{A}, \mathcal{S}_2) = \sum_{\langle s, a \rangle \in \llbracket \mathcal{S}_1, \mathcal{A} \rrbracket_{\mathcal{C}}} \sum_{s' \in \llbracket \mathcal{S}_2 \rrbracket_{\mathcal{C}}} w(s) \cdot T(s, a, s')$$

Here, $w(s)$ is a weighting factor that ensures $T_{\mathcal{C}}$ is well-defined (see Section 3.4). Having defined the transition probabilities between abstract states, the reward function is now defined in a similar way. Let \mathcal{S}_1 and \mathcal{S}_2 be abstract states in \mathcal{C} and let $\mathcal{A} \in A(\mathcal{S}_1)$.

$$R_{\mathcal{C}}(\mathcal{S}_1, \mathcal{A}, \mathcal{S}_2) = \sum_{\langle s, a \rangle \in \llbracket \mathcal{S}_1, \mathcal{A} \rrbracket_{\mathcal{C}}} \sum_{s' \in \llbracket \mathcal{S}_2 \rrbracket_{\mathcal{C}}} w(s) \cdot R(s, a, s')$$

In general, the decision problem is no longer guaranteed to be *Markovian* (see Chapter 2) at the abstraction level the CARCASS provides. Preserving this one-step model can be done by ensuring that the abstract state-action space implements a *stochastic bi-simulation* (see Section 3.4). But because a CARCASS also abstracts over actions, we need a different condition. As an illustration, see the following example.

Let $\mathcal{S} \equiv \text{on}(A, B), \text{on}(C, D), \text{on}(E, f)$ and consider the abstract actions $\mathcal{A}_1 \equiv \text{move}(A, C)$, $\mathcal{A}_2 \equiv \text{move}(C, A)$ and $\mathcal{A}_3 \equiv \text{move}(A, f)$ that are applicable in \mathcal{S} . Let us assume that \mathcal{S} is the first state in the ordering, and also, let

$$s \equiv \text{on}(a, b), \text{on}(b, c), \text{on}(c, f), \text{on}(d, e), \text{on}(e, f), \text{on}(g, f)$$

Now we can notice the following things: **i)** \mathcal{A}_1 and \mathcal{A}_2 both aggregate over $\text{move}(a, d)$ in s , i.e. $\text{move}(a, d) \in (\llbracket \mathcal{S}, \mathcal{A}_1 \rrbracket_{\mathcal{C}}^s \cap \llbracket \mathcal{S}, \mathcal{A}_2 \rrbracket_{\mathcal{C}}^s)$, and **ii)** \mathcal{A}_3 aggregates $\text{move}(a, \text{floor})$ and $\text{move}(d, \text{floor})$, i.e. $\{\text{move}(a, \text{floor}), \text{move}(d, \text{floor})\} \subseteq \llbracket \mathcal{S}, \mathcal{A}_3 \rrbracket_{\mathcal{C}}^s$. This shows how actions are dependent on the abstractions and the specific ground state. It shows that abstract actions can have many groundings, and that the same groundings can stem from different abstract actions. This alters the way we have to look at abstract transitions. The condition on abstract states and actions and - therefore on a CARCASS - is the following. Let \mathcal{S}_i and \mathcal{S}_j be two abstract states and let $M = \langle \mathcal{S}, \mathcal{A}, T, R \rangle$ be an RMDP. Let $s_1, s_2 \in \llbracket \mathcal{S}_i \rrbracket_{\mathcal{C}}$. The Markov property for the aggregated RMDP $M_{\mathcal{C}}$ is maintained if for any abstract state \mathcal{S}_j

$$\sum_{a \in \llbracket \mathcal{S}_i, \mathcal{A} \rrbracket_{\mathcal{C}}^{s_1}} \sum_{s' \in \llbracket \mathcal{S}_j \rrbracket_{\mathcal{C}}} T(s_1, a, s') = \sum_{a \in \llbracket \mathcal{S}_i, \mathcal{A} \rrbracket_{\mathcal{C}}^{s_2}} \sum_{s' \in \llbracket \mathcal{S}_j \rrbracket_{\mathcal{C}}} T(s_2, a, s') \quad (5.1)$$

This condition ensures that the dynamic behavior will be Markov, but it does not say anything about the rewards. Rewards for abstract transitions will be averaged over all the ground transitions that are aggregated. The abstraction level is a stochastic bi-simulation if, in addition to the requirement on the transition probabilities, for each two states $s_1, s_2 \in \llbracket \mathcal{S}_i \rrbracket_{\mathcal{C}}$ the rewards are the same for all abstract actions. The extra terms in Equation 5.1 that sum over the ground states aggregated by the abstract action \mathcal{A} are due to the relational setting in which an abstract action essentially represents a set of transitions ranging over a multiple ground actions. Whether a CARCASS implements a stochastic bi-simulation, or any other type of abstraction we have described in Section 3.4, is determined by the designer of the abstraction.

5.2.2 Q -Learning for CARCASSs

Because CARCASSs induce a partition over the state-action space, they provide the basis for abstract Q -value functions. The possibility of augmenting the state-action pairs in a CARCASS with values provides the basis for using any kind of model-free value function learning algorithm to learn a Q -value function for an RMDP.

DEFINITION 5.2.4 ▶ An **abstract Q -value function** for a CARCASS \mathcal{C} , denoted $Q_{\mathcal{C}}$, assigns to each abstract state-action pair in \mathcal{C} a value $Q_{\mathcal{C}}(\mathcal{S}_i, \mathcal{A}_{ij})$. For any specific RMDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$, $Q_{\mathcal{C}}$ assigns a value q to each state-action pair $\langle s, a \rangle$ (where $s \in \mathcal{S}$ and $a \in \mathcal{A}$) using

$$Q_{\mathcal{C}}(s, a) = Q_{\mathcal{C}}(\mathcal{S}_i, \mathcal{A}_{ij}) \text{ if } \langle s, a \rangle \in \llbracket \mathcal{S}_i, \mathcal{A}_{ij} \rrbracket_{\mathcal{C}}$$

In this way, $Q_{\mathcal{C}}$ implements an abstract Q -function where the states and actions aggregated by the CARCASS \mathcal{C} share their Q -value. It is important to note that by aggregating states and actions, we implicitly *average* their Q -values during learning. When the aggregation is not in correspondence with the underlying RMDP dynamics, problems involving *partial observability* can be expected, and a resulting policy may not be optimal in the ground RMDP. Similarly one can define an abstract policy based on the same semantics.

DEFINITION 5.2.5 ▶ An **abstract policy** is a CARCASS $\pi_{\mathcal{C}} = [\langle \mathcal{S}_i, \langle \mathcal{A}_i \rangle \rangle]$, i.e. in which $\#A_{\mathcal{C}}(\mathcal{S}_i) = 1$ for all abstract states \mathcal{S}_i in $\pi_{\mathcal{C}}$. For any specific RMDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$, $\pi_{\mathcal{C}}$ assigns an action $a \in \mathcal{A}$ to each state $s \in \mathcal{S}$ using

$$\pi_{\mathcal{C}}(s) = \text{choice}(\llbracket \mathcal{S}_i \rrbracket_{\pi_{\mathcal{C}}}^s)$$

Where *choice* selects an action randomly from the set of ground actions in $\llbracket \mathcal{S}_i \rrbracket_{\pi_{\mathcal{C}}}^s$.

The definitions of value functions and policies are special cases of the definitions in Section 4.5.1.1. For CARCASSs we use a strict ordering of the rules, ensuring that a proper partition is defined. Definition 5.2.5 indicates that if we can interpret a CARCASS in which $\#A_{\mathcal{C}}(\mathcal{S}_i) = 1$ for all abstract states \mathcal{S}_i as a policy, one can see a general CARCASS (i.e. where $\#A_{\mathcal{C}}(\mathcal{S}_i) > 1$) as a *bias on the policy space*. In other words, a CARCASS limits the number of available actions in each state through abstraction. Each abstract action in the action space $A_{\mathcal{C}}(\mathcal{S}_i)$ for some $\mathcal{S}_i \in \mathcal{C}$ can be seen as a possible course of action. The key idea now is to learn an action-value function for abstract actions in order to learn which one is best, and use that information to select the best policy. In essence every model-free RL technique can be used to learn the Q -values, such as Monte-Carlo methods, SARSA and $Q(\lambda)$ (see Chapter 2). Here we focus on standard Q -learning for our abstract representation and give a general algorithm.

A Q -learning method for CARCASSs is depicted in Algorithm 8. It follows the general outline of model-free RL algorithms as described in Chapter 2, but now adapted to the CARCASS representation and semantics. Let us assume the CARCASS from Example 5.2.1 and that the current state is

$$s \equiv (\text{on}(a, \text{floor}), \text{on}(b, \text{floor}), \text{on}(c, \text{floor}), \text{clear}(a), \text{clear}(b), \text{clear}(c))$$

The first abstract state that covers s is \mathcal{S}_2 . Assume that the Q -value for abstract action $\mathcal{A}_{21} \equiv \text{move}(A, B)$ is the highest and we choose this one to execute (i.e. no exploration).

Algorithm 8 Model-free Q -learning for CARCASSs.

Require: $\mathcal{C} = [\langle \mathcal{S}_i, \langle \mathcal{A}_{i1}, \dots, \mathcal{A}_{im_i} \rangle \rangle]$ is a CARCASS

Require: $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$ is an RMDP

Require: $Q_{\mathcal{C}}$ is a Q -value function for \mathcal{C}

```

1: for all episodes do
2:   initialize the start state  $s$ 
3:   while not ((end of episode) or (maximum number of steps reached)) do
4:      $s$  is current ground state
5:     find:  $\mathcal{S} \in \mathcal{C}$  for which  $s \in \llbracket \mathcal{S} \rrbracket_{\mathcal{C}}$ .
6:     if exploration then
7:       use exploration strategy to choose  $\mathcal{A}$  from  $A_{\mathcal{C}}(\mathcal{S})$ 
8:     else
9:        $\mathcal{A} = \arg \max_{\mathcal{A} \in A_{\mathcal{C}}(\mathcal{S})} Q_{\mathcal{C}}(\mathcal{S}, \mathcal{A})$ 
10:    choose a random action  $a \in \llbracket \mathcal{S}, \mathcal{A} \rrbracket_{\mathcal{C}}^s$ 
11:    observe new state  $s'$  and reward  $r$ 
12:    find new abstract state  $\mathcal{S}' \in \mathcal{C}$  for which  $s' \in \llbracket \mathcal{S}' \rrbracket_{\mathcal{C}}$ .
13:     $Q_{\mathcal{C}}(\mathcal{S}, \mathcal{A}) = Q_{\mathcal{C}}(\mathcal{S}, \mathcal{A}) + \alpha(r + \gamma \max_{\mathcal{A}' \in A_{\mathcal{C}}(\mathcal{S}')} Q_{\mathcal{C}}(\mathcal{S}', \mathcal{A}') - Q_{\mathcal{C}}(\mathcal{S}, \mathcal{A}))$ .
    
```

State s belongs to abstract state \mathcal{S}_2 because it is the first state in the ordering that has s as its model, i.e. $s \vdash \mathcal{S}$. This generates a number of possible substitutions for the variables A and B . Two of these possibilities are

$$\theta_1 = \{A/c, B/a\} \text{ and } \theta_2 = \{A/a, B/b\}.$$

Under the given abstraction, one cannot distinguish between these and a substitution is chosen at random. For example, one can take the action $\text{move}(c, a)$. After performing this action, a probabilistic transition is made to a new (abstract) state and a reward is received. Now one can update the Q -value of $\langle \mathcal{S}_2, \text{move}(A, B) \rangle$. The complete algorithm is depicted in Algorithm 8. After the values of the abstract Q -function $Q(\mathcal{S}, \mathcal{A})$ for all $\mathcal{S} \in \mathcal{C}$ and all $\mathcal{A} \in A_{\mathcal{C}}(\mathcal{S})$ have converged, an *abstract policy* can be obtained from $\mathcal{C} = [\langle \mathcal{S}_1, \langle \mathcal{A}_{11}, \dots, \mathcal{A}_{1m_1} \rangle \rangle, \dots, \langle \mathcal{S}_n, \langle \mathcal{A}_{n1}, \dots, \mathcal{A}_{nm_n} \rangle \rangle]$ by using $Q_{\mathcal{C}}$ in the following way

$$\pi_{\mathcal{C}} = [\langle \mathcal{S}_1, \langle \arg \max_{\mathcal{A}=\mathcal{A}_{11}, \dots, \mathcal{A}_{1m_1}} Q_{\mathcal{C}}(\mathcal{S}_1, \mathcal{A}) \rangle \rangle, \dots, \langle \mathcal{S}_n, \langle \arg \max_{\mathcal{A}=\mathcal{A}_{n1}, \dots, \mathcal{A}_{nm_n}} Q_{\mathcal{C}}(\mathcal{S}_n, \mathcal{A}) \rangle \rangle]$$

This amounts simply to taking the best abstract action for each abstract state (ties are broken randomly). The learning setting for CARCASSs can now be stated more explicitly. Let $\pi(\mathcal{C})$ be the set of all policies that can be constructed using the transformation we have stated in the equation above. The number of possible policies for a single CARCASS \mathcal{C} containing n abstract states is

$$\#\pi(\mathcal{C}) = \#A_{\mathcal{C}}(\mathcal{S}_1) \times \dots \times \#A_{\mathcal{C}}(\mathcal{S}_n)$$

Learning in the CARCASS setting amounts to *finding the best policy in the set* $\pi(\mathcal{C})$, choosing between $\#\pi(\mathcal{C})$ different policies. Whether the true optimal ground policy is modeled in this set, is part of the modeling process: the more refined the CARCASS is, the more probable this becomes. At the same time, a more refined CARCASS yields a larger representation, and more computational efforts involved in the learning process. This is yet

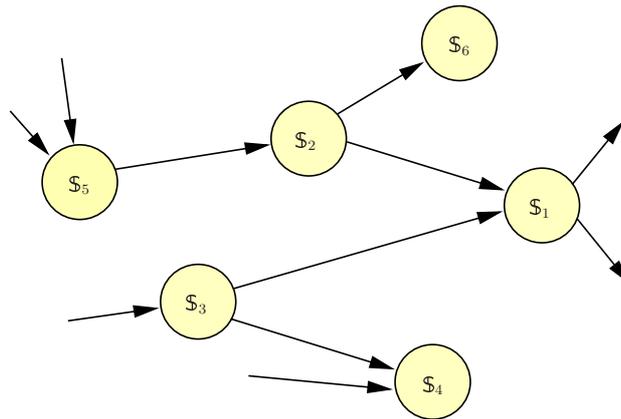


Figure 5.1: Part of the state transition graph of an RMDP.

another illustration of the general trade-off between the coarseness of the representational devices used, and the computational efforts and accuracy of the solution.

Note that the type of Q -learning we use for CARCASSs can be used to learn a Q -value function for any specific RMDP. But, as we know from Chapter 4, a value function is different for each ground RMDP. Q -learning on the abstract level will still converge if samples come from different RMDPs, but there are less guarantees that a resulting abstract value function (or policy) is optimal in each of the ground RMDPs. The policy that is extracted from the Q -value function learned in a specific RMDP, may be transferred to other RMDPs though (see the examples later in this section). Convergence of the Q -learning algorithm for CARCASSs is a direct consequence of the results for propositional state aggregations (e.g. Singh *et al.*, 1995) and the fact that a CARCASS is an *averager* (see Section 3.4).

5.2.3 Indirect Value Learning for CARCASSs using Approximate Models

Q -learning over an abstraction layer such as a CARCASS provides means to learn a value function and select the best policy, with respect to the abstraction level. However, by noticing that a CARCASS is not only useful to abstract over the Q -function, but also over the state-action space itself, we can make learning more efficient. Remember from Definition 5.2.3 that every CARCASS \mathcal{C} induces an abstract RMDP $M_{\mathcal{C}} = \langle S_{\mathcal{C}}, A_{\mathcal{C}}, T_{\mathcal{C}}, R_{\mathcal{C}} \rangle$. And even though in the model-free setting no *symbolic* transition model (e.g. in a STRIPS-like form) is available, we can still try to estimate the transition probabilities and the reward function over $M_{\mathcal{C}}$, i.e. the state transition graph such as depicted in Figure 5.1. Such an *approximate* model can be used for more efficient value function learning, in this case using *prioritized sweeping* (Moore and Atkeson, 1993, and see related algorithms in Section 2.6.3). Learning world models has wide applicability in RL (see also Chapter 2), and they can be used to perform *full backups* of values instead of the one-step, sample-based backups we have used in Q -learning. The estimated probabilities and associated rewards for transitions between abstract states can be calculated from the following counters that

Algorithm 9 Prioritized Sweeping (Moore and Atkeson, 1993) adapted for CARCASSs, using Wiering (1999)’s adaptation for the priority queue.

Require: $\mathcal{C} = [\langle \mathcal{S}_i, \langle \mathcal{A}_{i1}, \dots, \mathcal{A}_{imi} \rangle \rangle]$ is a CARCASS

Require: \mathcal{S} is the most recently visited abstract state

```

1: for all  $\mathcal{A} \in A_{\mathcal{C}}(\mathcal{S})$  do
2:    $Q_{\mathcal{C}}(\mathcal{S}, \mathcal{A}) := \sum_{\mathbb{T}} \bar{T}_{\mathcal{S}\mathbb{T}}(\mathcal{A})(\bar{R}_{\mathcal{S}\mathbb{T}}(\mathcal{A}) + \gamma V_{\mathcal{C}}(\mathbb{T}))$ 
3:   promote the most recent state to the top of  $PQ$ 
4:    $n := 0$ 
5: while  $(n < U_{\max}) \wedge (PQ \neq \emptyset)$  do
6:   remove the top state  $\mathcal{S}$  from  $PQ$ 
7:    $\delta(\mathcal{S}) := 0$ 
8:   for all predecessor states  $\mathbb{T}$  of  $\mathcal{S}$  do
9:      $V'_{\mathcal{C}}(\mathbb{T}) := V_{\mathcal{C}}(\mathbb{T})$ 
10:    for all  $\mathcal{A} \in A_{\mathcal{C}}(\mathbb{T})$  do
11:       $Q_{\mathcal{C}}(\mathbb{T}, \mathcal{A}) := \sum_{\mathbb{U}} \bar{T}_{\mathbb{T}\mathbb{U}}(\mathcal{A})(\bar{R}_{\mathbb{T}\mathbb{U}}(\mathcal{A}) + \gamma V_{\mathcal{C}}(\mathbb{U}))$ 
12:       $V_{\mathcal{C}}(\mathbb{T}) := \max_{\mathcal{A} \in A_{\mathcal{C}}(\mathbb{T})} Q_{\mathcal{C}}(\mathbb{T}, \mathcal{A})$ 
13:       $\delta(\mathbb{T}) := \delta(\mathbb{T}) + V_{\mathcal{C}}(\mathbb{T}) - V'_{\mathcal{C}}(\mathbb{T})$ 
14:      if  $|\delta(\mathbb{T})| > \epsilon$  then
15:        promote  $\mathbb{T}$  to priority  $|\delta(\mathbb{T})|$ 
16:       $n := n + 1$ 
17:    $PQ := \emptyset$ 

```

are updated after each action:

- $X_{\mathcal{S}\mathbb{T}}^{\mathcal{A}}$ = the number of transitions from abstract state \mathcal{S} to abstract state \mathbb{T} after executing the abstract action \mathcal{A}
- $X_{\mathcal{S}}^{\mathcal{A}}$ = the number of times the agent has executed the abstract action \mathcal{A} in state \mathcal{S}
- $R_{\mathcal{S}\mathbb{T}}^{\mathcal{A}}$ = the sum of rewards received by the agent by executing the action \mathcal{A} in state \mathcal{S} and making a transition to state \mathbb{T}

With these quantities an approximate transition and reward model over the CARCASS abstraction level can be calculated as follows:

$$\bar{T}_{\mathcal{S}\mathbb{T}}(\mathcal{A}) = \frac{X_{\mathcal{S}\mathbb{T}}^{\mathcal{A}}}{X_{\mathcal{S}}^{\mathcal{A}}} \quad \text{and} \quad \bar{R}_{\mathcal{S}\mathbb{T}}(\mathcal{A}) = \frac{R_{\mathcal{S}\mathbb{T}}^{\mathcal{A}}}{X_{\mathcal{S}}^{\mathcal{A}}} \quad (5.2)$$

In Algorithm 9 the *prioritized sweeping* (PS) method for updating the Q -values is shown. When using PS, line 13 in Algorithm 8 is replaced by a call to Algorithm 9. Whereas Q -learning only updates the Q -values along the experienced trace, PS updates many Q -values throughout the state-action space. By using the transition model updates for one state can be *propagated* to other states that have an action leading to that state. Note that we use a state value function $V_{\mathcal{C}}$ here that is defined similarly to $Q_{\mathcal{C}}$ (see Definition 5.2.4).

As an example, let us assume the most recent state visited is state \mathcal{S}_1 in Figure 5.1. First all Q -values of the actions in $A_{\mathcal{C}}(\mathcal{S}_1)$ are updated through the uses of the approximate model in line 2, and \mathcal{S}_1 is put at the top of a priority queue (PQ) in line 3. Then the algorithm performs a series of updates that use the approximate model to propagate the updates *backwards* through the state space, thereby using the size of the updates as a

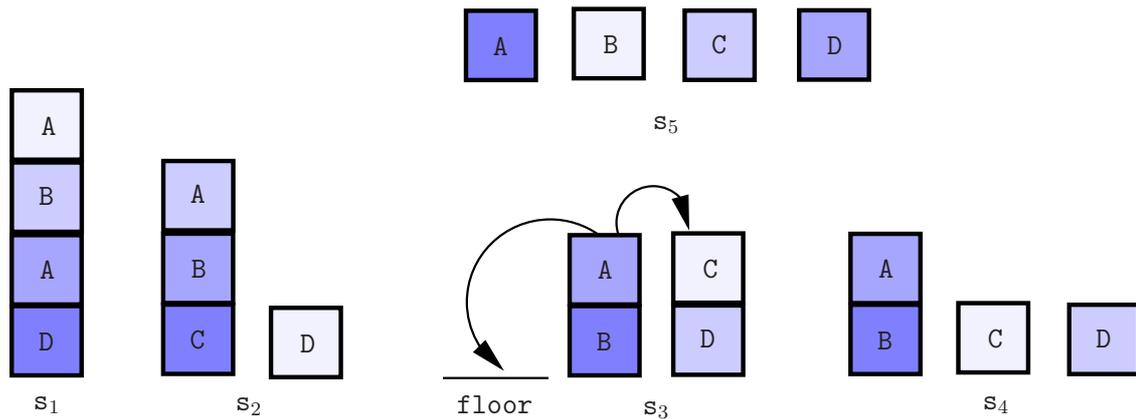


Figure 5.2: All block configurations in a BLOCKS WORLD consisting of four blocks. This reduces the number of 73 RMDP states to only 5 abstract ones, by abstracting over block names. In abstract state \mathcal{S}_3 the two available actions $\text{move}(A, C)$ and $\text{move}(A, \text{floor})$ are depicted, and omitted are the (equivalent, modulo substitutions) actions $\text{move}(C, A)$ and $\text{move}(C, \text{floor})$. For the other abstract states, applicable actions are omitted. Note that, when implemented as a CARCASS, it applies to BLOCKS WORLDS of arbitrary size, since there can be infinitely many blocks not mentioned in the specification.

heuristic for choosing whether other states will have to be updated too. For example, after updating the values for state \mathcal{S}_1 , the approximate model can be used to find the states \mathcal{S}_2 and \mathcal{S}_3 . Presumably, because their action values depend, in part, on the value of state \mathcal{S}_1 , and because this value has been updated, we may want to update the action values of \mathcal{S}_2 and \mathcal{S}_3 . If such an update would be large enough, a state is entered into the PQ (line 15). The maximum number of updates resulting from one step in the world is determined by the parameter U_{\max} , influencing a trade-off between sample complexity and computational complexity (see Chapter 2). Note that we use a small modification of the original PS algorithm, by altering the order of updating the value of a state and the insertion of that state in the PQ (see Wiering, 1999, p. 59–61).

5.2.4 Analysis and Experiments

In this section we will show results of some experiments and perform some basic analysis. We use a PROLOG⁴ implementation of CARCASSs, using the YAP⁵ PROLOG system (v.4.4.4). For all experiments, we reset the random number generator using YAP’s `randset` predicate (using numbers derived from `statistics(walltime, .)`) to get fully randomized experiments. All estimated numbers are obtained using a WINDOWS XP based system on a 1.1 GHz processor with 512 megabytes of RAM. In general, we focus on representational aspects and how they affect learning.

Experiment I: Abstracting from Block Names. The purpose of our first example is to show the benefits of abstraction. We use a CARCASS that is similar to the one in Figure 5.2 where a version for four blocks is depicted. The abstraction level abstracts away from individual blocks, and uses *configurations* instead. For example, state \mathcal{S}_3 covers all states in which the four blocks are in two stacks. Possible actions are to put a block from one

⁴Our original implementation was based on a combination of JAVA and the SICSTUS PROLOG system, through the JASPER interface. The version in the experiments is a full PROLOG version, tested in YAP.

⁵<http://sourceforge.net/projects/yap>

stack on top of the other stack (resulting in a transition to configuration s_2) and to put a block from a stack onto the floor (resulting in a transition to configuration s_4). Such an abstraction level effectively decreases the number of states from 73 ground states to only 5 abstract states. We use an additional PROLOG definition of the background predicate $\text{nrTowers}(N)$ that counts the number of blocks on the floor (which equals the number of towers in the current state). This predicate makes it easier to specify the configurations.

$$\text{nrTowers}(N) : - \text{findall}(X, \text{on}(X, \text{floor}), Xs), \text{length}(Xs, N).$$

For the experiments we use the same type of abstraction, now adapted to five blocks, modeled as the following CARCASS \mathcal{C}_{BW5} :

state (\mathcal{S}_1):	$(\text{nrTowers}(1), \text{cl}(A), A \neq \text{floor})$
actions ($\mathcal{A}_{11}, \dots, \mathcal{A}_{11}$):	$\text{move}(A, \text{floor})$
state (\mathcal{S}_2):	$(\text{nrTowers}(2), \text{cl}(A), \text{on}(A, \text{floor}), \text{cl}(B), B \neq \text{floor}, A \neq B)$
actions ($\mathcal{A}_{21}, \dots, \mathcal{A}_{23}$):	$\text{move}(A, B), \text{move}(B, A), \text{move}(B, \text{floor})$
state (\mathcal{S}_3):	$(\text{nrTowers}(2), \text{cl}(A), \text{on}(A, B), \text{on}(B, \text{floor}), \text{cl}(C),$ $C \neq A, C \neq \text{floor})$
actions ($\mathcal{A}_{31}, \dots, \mathcal{A}_{34}$):	$\text{move}(A, C), \text{move}(C, A), \text{move}(C, \text{floor}), \text{move}(A, \text{floor})$
state (\mathcal{S}_4):	$(\text{nrTowers}(3), \text{cl}(A), \text{cl}(B), \text{on}(A, \text{floor}), \text{on}(B, \text{floor}),$ $A \neq B, \text{cl}(C), C \neq \text{floor}, A \neq C, B \neq C)$
actions ($\mathcal{A}_{41}, \dots, \mathcal{A}_{47}$):	$\text{move}(A, C), \text{move}(B, C), \text{move}(C, A), \text{move}(C, B),$ $\text{move}(A, B), \text{move}(B, A), \text{move}(C, \text{floor})$
state (\mathcal{S}_5):	$(\text{nrTowers}(3), \text{cl}(A), \text{cl}(B), A \neq B, \text{on}(C, \text{floor}),$ $\text{cl}(C), C \neq A, C \neq B, B \neq \text{floor}, A \neq \text{floor})$
actions ($\mathcal{A}_{51}, \dots, \mathcal{A}_{58}$):	$\text{move}(A, B), \text{move}(A, C), \text{move}(A, \text{floor}), \text{move}(B, A),$ $\text{move}(B, C), \text{move}(B, \text{floor}), \text{move}(C, A), \text{move}(C, B)$
state (\mathcal{S}_6):	$(\text{nrTowers}(4), \text{on}(A, B), \text{on}(B, \text{floor}), \text{cl}(A), \text{cl}(C), A \neq C, C \neq \text{floor})$
actions ($\mathcal{A}_{61}, \dots, \mathcal{A}_{63}$):	$\text{move}(A, \text{floor}), \text{move}(A, C), \text{move}(C, A)$
state (\mathcal{S}_7):	$(\text{nrTowers}(5), \text{cl}(A), \text{cl}(B), A \neq B, A \neq \text{floor}, B \neq \text{floor})$
actions ($\mathcal{A}_{71}, \dots, \mathcal{A}_{71}$):	$\text{move}(A, B)$

So, although we still need as many abstract states as there are configurations, the number of states is considerably decreased (from 501 to 7) and the total number of state-action pairs is only 27. Note that this representation still contains unnecessary redundancies⁶ For example, in state \mathcal{S}_4 all actions concerning blocks A and B are effectively the same, because they will result in the same substitutions. Other redundancies could be removed by making use of the rule order. For example, the test $\text{nrTowers}(5)$ in state \mathcal{S}_7 is unnecessary because when we arrive at this state, we know that all the blocks are on the floor. Nevertheless, we keep the conceptually more clear representation because it is more close to the concept of the configurations. Because \mathcal{C}_{BW5} just abstracts from block names, the transitions are Markov at this level of abstraction.

In our experiments we use a BLOCKS WORLD containing 5 blocks, one version in which actions are deterministic, and one in which all actions can fail with probability 0.1. The goal is to stack all blocks, i.e. to reach state \mathcal{S}_1 (resulting in a reward +1). For each episode, we set a maximum of 8 steps. If the goal is not reached by that time a new

⁶Depending on the implementation of the environment, some atoms are unnecessary for another reason. For example, enforcing that for an action $\text{move}(A, B)$ the first parameter A cannot be a `floor`, might be resolved elsewhere, for example when generating the substitutions. Here we simply list all constraints.

episode starts. Both algorithms use a fixed 0.1 exploration probability and $\alpha = 0.1$ for Q -learning and $U_{\max} = 10$ for PS. We omit learning curves here, because both Q -learning and PS compute optimal policies after only a dozen of episodes. Even though the ground RMDP contains 501 states, $\mathcal{C}_{\text{BW}_5}$ only has seven states and a number of optimal policies. Each optimal move must increase the height of an existing highest stack by moving a block on top of it, and there is more than one way to ensure that. The resulting learning problem itself is therefore very simple, due to relational abstraction. The same abstraction level can be used to learn an unstack task (i.e. to reach the last abstract state). Note that any resulting policy can be deterministic on the abstraction level, though will be nondeterministic in the ground RMDP.

What is interesting to note are the differences between the two algorithms. Both learn optimal policies, but PS is more efficient⁷. Two important parameters are the number of steps allowed in an episode and the amount of exploration. One extreme case is to run PS with 1.0 probability of exploration. In this case, a true empirical model of the CARCASS is built, and learning essentially becomes a form of *asynchronous* DP using the estimated model. In the deterministic setting, this results in all optimal actions having exactly the same Q -values. In the probabilistic setting, small variances in (estimated) transition probabilities result in small preferences for actions over other (equally optimal) actions. For example, in the deterministic setting PS will deem actions \mathbb{A}_{51} , \mathbb{A}_{54} , \mathbb{A}_{57} and \mathbb{A}_{58} optimal, though in the probabilistic setting it will favor one of them at each point in time. Q -learning takes slightly longer to learn, because propagating the end reward to all actions takes longer. Both algorithms are affected by the amount of exploration. In the deterministic setting, PS only needs to try out each action once, since it uses full backups using the (deterministic) transition model. Q -learning needs to visit state-action pairs more than once, depending on the learning parameter α , in order to ensure that the Q -values converge to their optimal values. In the probabilistic setting, exploration is somewhat more relevant. Depending on the amount of exploration in both algorithms, it may take some time to try out all actions, though given the relatively small state space of $\mathcal{C}_{\text{BW}_5}$, differences are minor. Additionally, the more steps are allowed in one episode, the more likely it is that the goal state is reached and useful learning samples are obtained.

A Note on the Generation of Initial States. As we have described, the transition probabilities and rewards are dependent on the distribution of states generated by the policy. As a consequence, the *initial state distribution* is equally important. In general, generating truly random BLOCKS WORLD states according to the underlying probability distribution is a non-trivial problem. Section 4.1.2 has introduced the BLOCKS WORLD and ways to count the number of (and form) of the state space. The same principles can be used to generate⁸ BLOCKS WORLD states (see Slaney, 1995; Slaney and Thiébaux, 2001). In our experiments we opt for a more practical approach of generating initial states by sequentially and randomly placing all the blocks (either on the floor or on each other). Furthermore, we make

⁷But also more computationally expensive. Simulating 1000 episodes takes 2.25 seconds for Q -learning, but 2.9 seconds for PS (estimated with the YAP built-in predicate `statistics(cputime, .)`). Most other results on comparisons between Q -learning and PS apply here (see further Moore and Atkeson, 1993; Wiering, 1999). Compared to learning in the ground RMDP, both algorithms take only a fraction of the time to learn an optimal policy.

⁸Slaney (1995) reports on a publicly available program to generate random BLOCKS WORLD states of arbitrary size.

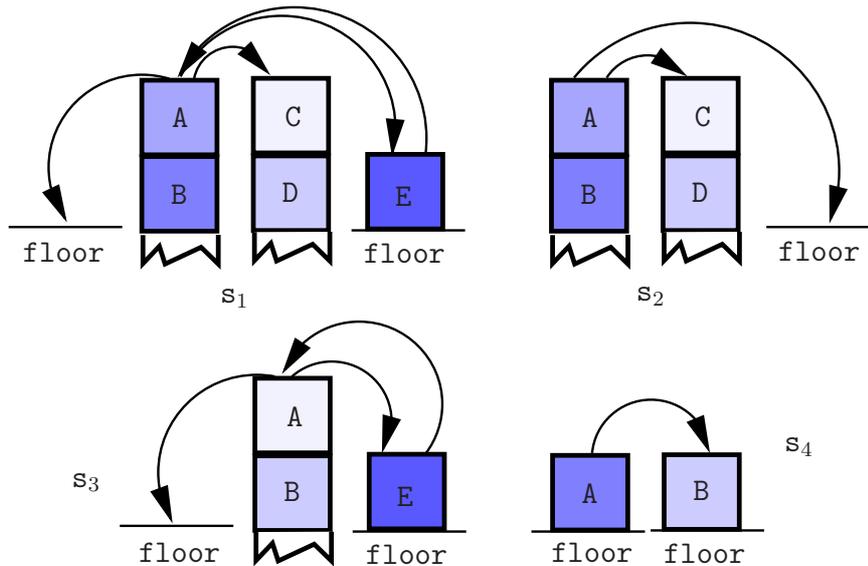


Figure 5.3: The CARCASS \mathcal{C}_{GBW} for BLOCKS WORLDS of arbitrary size. This CARCASS is useful for learning a stack behavior in which the task is to stack all blocks in the current BLOCKS WORLD. See the text for analysis and explanation.

sure that initial states are never goal states. The consequences for our results are minor, given the relatively small numbers of episodes compared to state space sizes (e.g. 10 blocks generate a state space of almost 60 million states).

Experiment II: Stacking in Arbitrary BLOCKS WORLDS. The goal of this experiment is to highlight some of the opportunities and caveats when using abstraction over RMDPs. Here we focus on an abstraction level that is useful for learning a stack behavior that carries over to arbitrary BLOCKS WORLDS of any size. Such *generalized policies* are described by Martin and Geffner (2000, 2004). A general strategy to reach a certain configuration of blocks (e.g. a stack position) is to use a two-phase strategy: **i)** first break down all existing stacks by placing each individual block on the floor, and then **ii)** build up the state that is required block by block. This may not yield very efficient policies, but they can be very simple, deterministic, and above all, they are independent of the number of blocks.

In Figure 5.3 we see a general CARCASS \mathcal{C}_2 that can be used to learn a stack behavior. This type of abstraction level was also used by Kersting and De Raedt (2004). This CARCASS applies to blocks worlds of arbitrary size. The abstraction level is coarse; only four abstract states are identified, though it may be applied to any specific RMDP (from 5 blocks up). Each abstract state describes a configuration of blocks that can be detected in the current RMDP state, and actions can be performed on the mentioned blocks. For example, in the second state one can move either a block from one stack to another or one of the top blocks can be moved to the floor. Note that in the analysis and experiments, we will use all possible actions (e.g. we further allow in this state the actions $\text{move}(C, A)$ and $\text{move}(C, \text{floor})$). We will now first analyze this abstraction level.

Analysis and Experiments. The CARCASS \mathcal{C}_{GBW} seems a suitable candidate for learning a generalized strategy for the stack task: use states \mathcal{S}_1 and \mathcal{S}_2 for moving all blocks to the floor and states \mathcal{S}_3 and \mathcal{S}_4 to build one large stack. However, there are subtle issues to address here. As an example, consider the first abstract state (\mathcal{S}_1) in Figure 5.3. It covers

all states in which there are at least two towers of height > 2 and at least one tower of height 1. The action $\text{move}(A, C)$ moves the top of the first tower to the top of the second, whereas the action $\text{move}(A, \text{floor})$ puts it on the floor. Intuitively these actions do very different things. We can show however that - *due to the abstraction level that is used* - for an RL algorithm, both appear the same. All states $s \in \llbracket \mathcal{S}_1 \rrbracket_{\mathcal{C}}$ are states with the following structure, e.g. a list of tower heights

$$\langle (2 + x), (2 + y), 1, t_1 \dots t_k \rangle$$

Basically, this covers three different types of situations:

1. $x = 0$ and $t_i = 1, \forall i = 1 \dots k$: then both actions $\text{move}(A, \text{floor})$ and $\text{move}(A, C)$ will decrease the height of the first tower to 1, so there will be only one tower left and we make a transition to \mathcal{S}_3
2. $x = 0$ and $\exists i t_i > 1$. In this case, the first tower disappears, but two towers of height > 1 will be left and both actions result in a transition to \mathcal{S}_1
3. $x > 0$: Both actions will result in a transition to state \mathcal{S}_1 because basically the same configuration is left, except that the first tower has its height decreased by one.

With this we can conclude⁹ that intuitively the two actions $\text{move}(A, \text{floor})$ and $\text{move}(A, C)$ in \mathcal{S}_1 are quite different, but their transition probabilities are exactly the same and hence they will get the same Q -value and therefore both actions are equally 'good'. An optimal policy can choose either one. By using equation 5.1, one can demonstrate that \mathcal{C}_{GBW} induces a non-Markovian abstraction, i.e. the outcome of actions *does* depend on the history of actions. For example, take the following notions (in a six-block world, where we omit clear):

$$\begin{aligned} s_1 &= \{\text{on}(a, b), \text{on}(b, c), \text{on}(c, f), \text{on}(d, e), \text{on}(e, f), \text{on}(g, f)\} \\ s_2 &= \{\text{on}(a, b), \text{on}(b, f), \text{on}(c, d), \text{on}(d, f), \text{on}(e, f), \text{on}(g, f)\} \end{aligned}$$

Now:

$$\begin{aligned} \llbracket \mathcal{S}_1, \mathcal{A}_1 \rrbracket_{\mathcal{C}}^{s_1} &= \{\text{move}(a, f), \text{move}(d, f)\} =_{\text{def}} \{a_{11}, a_{12}\} \\ \llbracket \mathcal{S}_1, \mathcal{A}_1 \rrbracket_{\mathcal{C}}^{s_2} &= \{\text{move}(a, f), \text{move}(c, f)\} =_{\text{def}} \{a_{21}, a_{22}\} \end{aligned}$$

Executing actions a_{i1} and a_{i2} in state s_i results in:

$$\begin{aligned} s_{11} &= \{\text{on}(b, c), \text{on}(c, f), \text{on}(d, e), \text{on}(e, f), \text{on}(a, f), \text{on}(g, f)\} \\ s_{12} &= \{\text{on}(a, b), \text{on}(b, c), \text{on}(c, f), \text{on}(d, f), \text{on}(e, f), \text{on}(g, f)\} \\ s_{21} &= \{\text{on}(c, d), \text{on}(a, f), \text{on}(b, f), \text{on}(e, f), \text{on}(g, f)\} \\ s_{22} &= \{\text{on}(a, b), \text{on}(c, f), \text{on}(d, f), \text{on}(e, f), \text{on}(g, f)\} \end{aligned}$$

And now, according to equation 5.1, we can see that the Markov property does not hold for this abstract state space and that will result in problems concerning partial observability and the use of value function learning methods that are based on this Markov property.

$$T(s_1, a_{11}, s_{12}) \neq T(s_2, a_{21}, s_{21}) + T(s_2, a_{22}, s_{22})$$

Because $0.5 \neq 1.0$.

By letting the PS algorithm run with 1.0 exploration probability, we obtain an empirical validation of this aspect. Here we use 8 blocks, and the maximum number of steps per episode is 15. Let AC stand for $\text{move}(A, C)$ and Af for $\text{move}(A, \text{floor})$

⁹A similar analysis applies to state \mathcal{S}_2 if we delete state \mathcal{S}_1 from \mathcal{C}_{GBW} .

# ep	$T(\mathcal{S}_1, Af, \mathcal{S}_1)$	$T(\mathcal{S}_1, AC, \mathcal{S}_1)$	$T(\mathcal{S}_1, Af, \mathcal{S}_3)$	$T(\mathcal{S}_1, AC, \mathcal{S}_3)$	time(s)
5000	6448 (0.674)	6492 (0.680)	3113 (0.326)	3056 (0.320)	135
10000	12963 (0.678)	13153 (0.678)	6158 (0.322)	6246 (0.322)	274

Here we can see that the empirical model converges to $\text{move}(A, C)$ and $\text{move}(A, \text{floor})$ having the same (empirical) transition probabilities. Because a learning algorithm cannot distinguish between these two actions, we first analyze the consequences for the resulting policies. In the following we have simulated the two different policies (in 1000 episodes) that result when we transform the CARCASS \mathcal{C}_{GBW} by taking the optimal actions $\text{move}(E, A)$ in state \mathcal{S}_3 and $\text{move}(A, B)$ in state \mathcal{S}_4 .

blocks	max steps	$\pi(\mathcal{S}_1)$	$\pi(\mathcal{S}_2)$	success	avg steps
8	20	$\text{move}(A, C)$	$\text{move}(A, \text{floor})$	0.994	6.667
		$\text{move}(A, \text{floor})$	$\text{move}(A, \text{floor})$	1.000	7.187
12	25	$\text{move}(A, C)$	$\text{move}(A, \text{floor})$	0.773	16.664
		$\text{move}(A, \text{floor})$	$\text{move}(A, \text{floor})$	1.000	13.784
15	30	$\text{move}(A, C)$	$\text{move}(A, \text{floor})$	0.523	24.899
		$\text{move}(A, \text{floor})$	$\text{move}(A, \text{floor})$	1.000	18.938

Apparently, the action $\text{move}(A, C)$ in state \mathcal{S}_1 makes the policy non-optimal, and its performance decreases when the number of blocks is increased. Finally, let us analyze what happens with the transition probabilities of actions $\text{move}(A, C)$ and $\text{move}(A, \text{floor})$ when we take either $\text{move}(A, C)$ or $\text{move}(A, \text{floor})$ in the first state. (We simulate 5000 episodes using PS with 1.0 probability of exploration, using 8 blocks and a maximum of 20 steps per episode). Let AC stand for $\text{move}(A, C)$ and Af for $\text{move}(A, \text{floor})$

π	$T(\mathcal{S}_2, AC, \mathcal{S}_1)$	$T(\mathcal{S}_2, AC, \mathcal{S}_2)$	$T(\mathcal{S}_2, AC, \mathcal{S}_3)$	$T(\mathcal{S}_2, Af, \mathcal{S}_1)$	$T(\mathcal{S}_2, Af, \mathcal{S}_3)$
AC	286 (0.146)	1288 (0.657)	385 (0.197)	1540 (0.800)	384 (0.200)
Af	283 (0.224)	806 (0.639)	173 (0.137)	1114 (0.895)	130 (0.105)

Experiments using both Q -learning and PS showed that if in the first state the correct action is chosen, the final policy is the generalized unstack–stack policy in which first all towers (except one) are broken down, after which one stack is built. However, if the suboptimal action $\text{move}(A, C)$ is chosen, the resulting policy is not optimal, with decreasing success probability in larger BLOCKS WORLDS. Both algorithms converge quickly because the space on which the actual learning is performed is very small (only four states), regardless of the number of blocks involved. However, learning times increase with the number of blocks, both because of the generation of initial states, and the increased length of learning and solution paths.

Experiment II: Learning to Play TIC-TAC-TOE. Our BLOCKS WORLD experiments have shown some of the properties of CARCASSs and algorithms for learning policies. Although BLOCKS WORLDS can have huge state spaces, due to the relational abstraction, the actual learning problems have been relatively simple. In our last experiment we show some of the benefits of CARCASS for a more challenging problem. In this experiment we use CARCASSs to learn reasonable policies for the game of TIC-TAC-TOE. Although a simple game, TIC-TAC-TOE consists of around 6000 ground states. In the introduction to Chapter 3 we have seen that this game provides many opportunities for abstraction, and a relational representation can make elegant use of that. Our aim is to beat a reasonable opponent. This opponent has only three rules: **i)** play on a square if you can win immediately, **ii)** if the first rule does not apply, but you can block your opponent from winning immediately, then do this, **iii)** otherwise, do a random move. Optimal play against this opponent yields a score of 0.614 on a scale from -1 (always lose) to $+1$ (always win) (Wiering, 1995). Our aim is not to create such an optimal player, but instead, we show that if some domain knowledge about the game is available, one can learn to play quite well against this opponent with modest modeling and learning efforts. Rewards are provided at the end of the game: 1 for winning, -1 for losing and 0 for a draw. The learning player plays \otimes and the reasonable opponent plays \odot . Each player has equal probability to start the game.

The CARCASS \mathcal{C}_{TTT} we use, is depicted in Figure 5.4. It models the decisions the learner can take for some of its first two moves, and furthermore it captures some additional positions. The abstract states are modeled using simple relations on the board, such as which positions are occupied by which player, and the concept of a *line* on the board. Each different variable in the figure stands for a different move. If multiple grid positions in one board have the same variable, they are similar actions. Note for that abstract states that are annotated with a $(*)$, the figure gives an example position, whereas all other positions show the exact abstract state. All states model those states in which our learner has to make a decision. Position $\$0$ covers all *goal* positions in which either one of the players has won, or the game is a draw. Position $\$1$ covers the situation where the learner has to move first. Positions $\$2$ – $\$4$ cover the positions in which the opponent has played the first move. Positions $\$5$ – $\$11$ cover those positions in which both players have made one move and it is the learner’s task to pick a move. Both positions $\$6$ and $\$8$ are examples of a general template. For state $\$6$ we know that \otimes is in a corner, and that \odot is not in the center square. Here, the learner can either **i)** move to a position on one of the lines \otimes is on, **ii)** move to a position that is not on a line in **i)** but is on the same line as \odot , and **iii)** move on a line that is occupied by neither \otimes nor \odot . A similar choice is offered in state $\$8$. State $\$12$ describes the situations in which the learner can win in one move, $\$13$ describes situations in which the opponent can win in one move, and $\$14$ describes possible fork-positions for the learner. In addition, there is a *catch-all* rule (modeled as state $\$15$) that covers all other positions and the only possible action here is to play at a randomly chosen empty grid position.

A fork-position is one where a player can make two lines, both of which can be won in a subsequent move. The opponent can only block one of these lines, thus it is a sure win (see the 7th and 8th positions in Figure 5.4). It is defined as

$$\text{forkX}(X) : - \text{emp}(X), \text{line}(L1), \text{line}(L2), \text{member}(X, L1), \text{member}(X, L2), \\ L1 \neq L2, \text{nX}(L1, 1), \text{n0}(L1, 0), \text{nX}(L2, 1), \text{n0}(L2, 0)$$

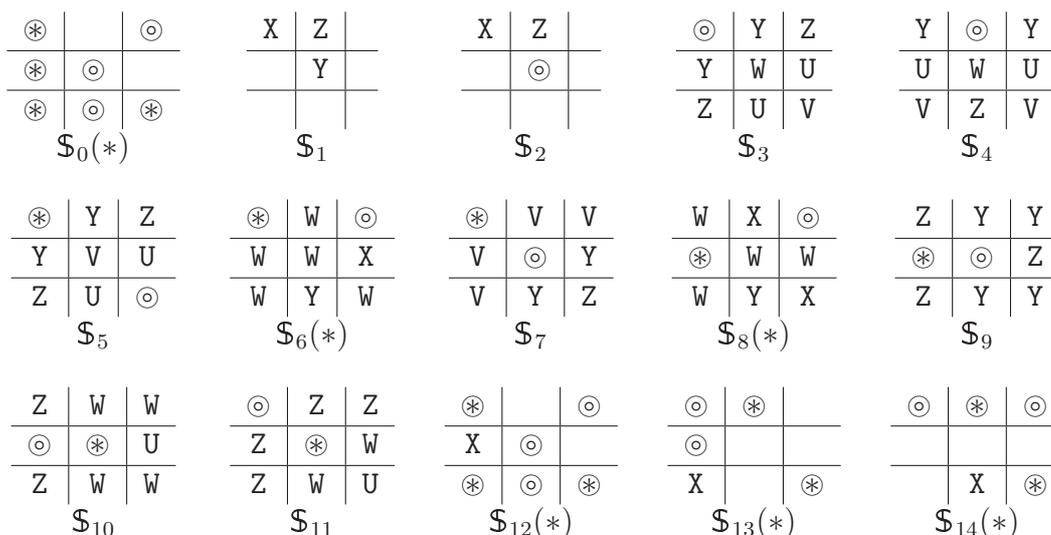
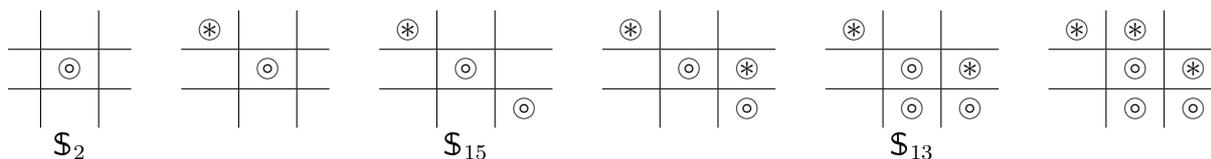


Figure 5.4: Abstract TIC-TAC-TOE Positions. The CARCASS \mathcal{C}_{TTC} . Note that the relational abstraction capabilities enable each abstract state to simultaneously cover all symmetrically equivalent boards, for example obtained by rotating the boards.

where n_X (and n_0) counts the number of X's (and 0's) on a line. The REASONABLE player cannot detect these fork positions, such that our player has an advantage. However, \mathcal{C}_{TTC} does not give the learner the opportunity to detect a fork-position for his opponent. Despite considerable success against the reasonable opponent, learned policies using \mathcal{C}_{TTC} can still be beaten if the opponent manages to (accidentally) create such a position. The following example game shows such a combination of moves that results in a loss.



And finally we arrive at \mathcal{S}_0 and receive a reward -1 .

Now let us take a look at some of the results of our experiments using \mathcal{C}_{TTC} . Figure 5.5 shows two typical learning curves for Q -learning and PS on the CARCASS \mathcal{C}_{TTC} . Each datapoint is the performance of the policies tested on 500 games. Each graph is an average over 5 experiments. For Q -learning we used the parameters $\epsilon = 0.05$, $\alpha = 0.01$ and $\gamma = 0.9$. For PS we used $\delta = 0.01$, $\gamma = 0.9$ and $\epsilon = 0.05$. Both learners achieve a performance estimate of around 0.4. Note that both the policy used by the learner (as well as the one used by the opponent) is stochastic in the ground RMDP, generating considerable stochasticity in the results.

Small differences in performance are attributable to the stochasticity in learning and testing, and the fact that there are several policies that differ only slightly in their Q -values but will generate small fluctuations in performance when tested. PS seems more vulnerable to this type of fluctuations, because updates tend to spread out more than in Q -learning. Policies derived during the Q -learning experiments are slightly more stable; once an action is deemed best, it becomes less likely to switch to another action with similar

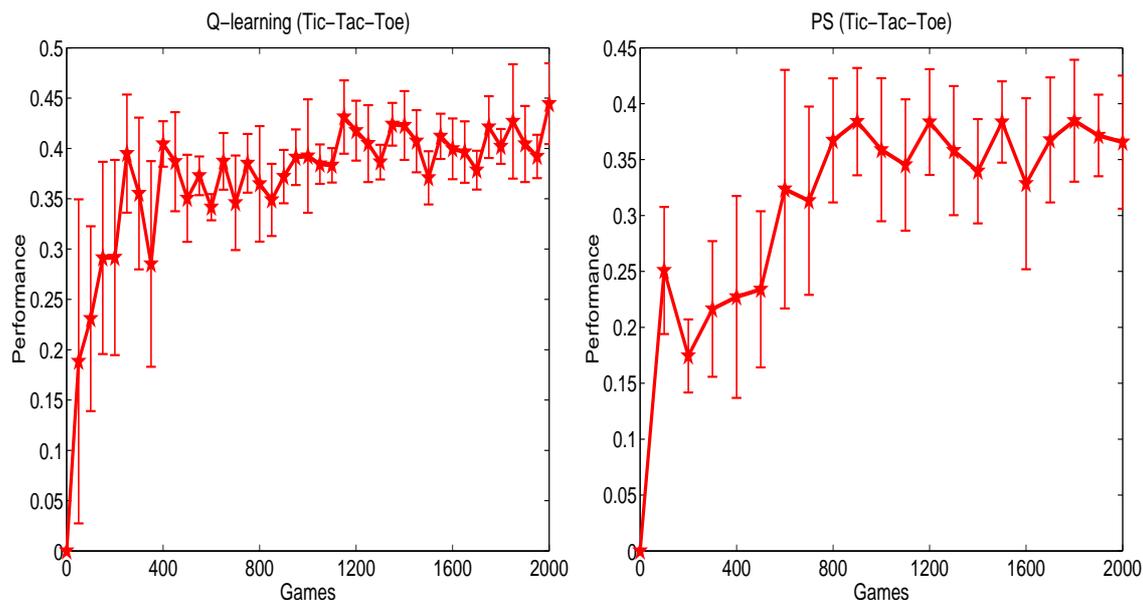


Figure 5.5: Results of Q -learning and PS on the CARCASS \mathcal{C}_{TTT} against the reasonable opponent. Each datapoint represents the performance when tested on 500 games. Note that we have only plotted one out of five datapoints for readability, i.e. the learners were tested after every 10 learning episodes.

Q -value. Changing the exploration mechanism to Boltzmann exploration (see Chapter 2) would change this. Interesting to note is that PS detects that $\text{move}(V)$ and $\text{move}(Z)$ in state \mathcal{S}_7 both have the same transition probabilities; both lead to a state where \odot can make a winning move. Q -learning will choose either one of them, depending on the particular preference at a certain time point.

Finally we compare our results against some other policies. We have created four different players (i.e. policies) and we have tested them against our reasonable opponent. A **random** player is modeled by the CARCASS consisting of only abstract states \mathcal{S}_0 and \mathcal{S}_{15} , i.e. each move is randomly selected from all empty grid positions. The **reasonable** opponent consists of states \mathcal{S}_0 , \mathcal{S}_{12} , \mathcal{S}_{13} , and \mathcal{S}_{15} . In order to find out how much benefit might be expected from the ability to detect *fork*-positions, we created the **fork** player that is equal to the **reasonable** player, except that we insert abstract state \mathcal{S}_{14} between \mathcal{S}_{13} and \mathcal{S}_{15} . Finally, our **learned** player is one of the best policies we encountered during a large batch of experiments. Using the variables in Figure 5.4, its policy chooses the following moves (starting from \mathcal{S}_1): $\langle Y, X, U, U, Z, Y, V \text{ or } Z, Y, Y, Z, U, X, X, X \rangle$. We tested all these players against the reasonable opponent and the results are shown in the following table. All numbers represent the averages over 10 runs of 10000 games played against our reasonable opponent.

	random	reasonable	fork	learned (\mathcal{C}_{TTT})
performance	-0.762 ± 0.005	0.00092 ± 0.006	0.165 ± 0.006	0.4143 ± 0.006
time(s)	22.51 ± 0.60	33.09 ± 1.59	44.68 ± 3.54	36.82 ± 1.54

Randomly selecting actions is not a good policy, and playing against itself results in a 0 score. But, the ability to detect *fork*-moves only gives a small performance increase. Our CARCASS \mathcal{C}_{TTT} performs quite well, given the fact that 0.614 is the maximum performance achievable. By only using some simple knowledge about the game and about symmetry

in the first two moves, we obtain a simple state space consisting of 16 abstract states (and 40 state-action pairs) that performs very well against the reasonable opponent. Remember that the ground state space consists of around 6000 states. The freedom to choose between opening moves pays off, and lets the learner increase its probability of generating a fork position. The message is clear; some additional domain knowledge can greatly increase performance in a task, while keeping representational and computational efforts modest.

5.2.5 Discussion

CARCASSs have been shown to be useful for abstracting over RMDPs. The three interpretations of a CARCASS, i.e. an abstraction over value functions, policies or state-action spaces, offer many opportunities for both modeling and learning. The BLOCKS WORLD problems have shown benefits and caveats of the relational abstraction in terms of the transition probabilities for the abstract states. The TIC-TAC-TOE example has shown an elegant use of CARCASS to create a small, comprehensible state-action space that is still powerful enough to learn good policies. An alternative would be to use so-called *after-states* in TIC-TAC-TOE. Using after-states amounts to learning values for the boards that are the result of some action. In this way an additional compression could be accomplished because it is often the case that many different moves result in a similar board position (see Sutton and Barto, 1998, for examples).

One relevant aspect has shown to be the size of the world, especially in the learning process but also in execution of policies. In BLOCKS WORLDS the number of blocks typically increases the average amount of steps needed to get to the goal position. For some of our experiments we have used 20 blocks but most often the learner never got off the ground because the required end reward for reaching the goal state at least once, was often never obtained. Luckily, the abstractions we have used can be used to learn in small BLOCKS WORLDS and resulting policies can be *transferred* to larger worlds. Still, this is an artefact of the BLOCKS WORLD and the tasks we have used. In general, the size of the world is an important factor that must be dealt with separately. For the initial exploration process, however, one solution would be to *guide* the learner towards the goal, for example by *shaping* (see Chapter 2) or by using a *guidance* policy, if available (e.g. see Driessens and Džeroski, 2004). The partial observability resulting from abstraction, and the problems concerning the counter-intuitive, equivalent transition probabilities of some of the abstract actions (e.g. see experiment II) are generic problems and must be dealt with in all approaches that employ some kind of relational abstraction.

One last aspect is the use of learning algorithms such as Q -learning and PS. Although they can provide useful means to find good policies, in some situations a simple alternative is available. If the number of state-action pairs is relatively small, one could use a brute-force search and test *all* possible policies and find out which one is best. Due to a fixed abstraction level, the policy space is fixed, and usually small. In our second experiment, such an approach only has to test a very small number of policies, and in fact, we have already almost done that when finding out the difference between the two equivalent actions in the first abstract state. In the second half of this chapter we use a similar approach in which we test policies holistically, although there the abstraction level is not fixed, and the resulting policy space is much larger (even infinite).

5.3. A Survey of Model-Free, Value-Based Approaches

In this section we survey existing techniques for VFA in first-order domains. As we have said, most of the algorithmics of the propositional setting (see Section 3.6) applies in the first-order setting, though generalization will now be performed using techniques from ILP (see Section 4.3.2). A large majority of the work in relational RL samples the underlying RMDP to *estimate* values for the current structures and uses this information for the induction of new structures or the modification of current ones. Remember from Chapter 4 (Equation 4.10) the following typical Q -learning pattern:

$$\begin{array}{ccccccc}
 \tilde{Q}^0 & \xrightarrow{\mathbf{S}} & \{\langle s, a, q \rangle\} & \xrightarrow{\mathbf{I}} & \tilde{Q}^1 & \xrightarrow{\mathbf{S}} & \{\langle s, a, q \rangle\} & \xrightarrow{\mathbf{I}} & \tilde{Q}^2 & \xrightarrow{\mathbf{S}} & \dots \\
 \downarrow \mathbf{D/I} & & \parallel & & \downarrow \mathbf{D/I} & & \parallel & & \downarrow \mathbf{D/I} & & \\
 \tilde{\Pi}^0 & \longrightarrow & \{\langle s, a, q \rangle\} & & \tilde{\Pi}^0 & \longrightarrow & \{\langle s, a, q \rangle\} & & \tilde{\Pi}^0 & \longrightarrow & \{\langle s, a, q \rangle\}
 \end{array}$$

Here, an initial abstract Q -function is used to get biased learning samples from the environment. The samples are then used to induce a new Q -function structure. A restricted variation on this scheme is to fix the logical abstraction level (e.g. \tilde{Q}) and only sample the environment to get good estimates of the values (e.g. parameters of \tilde{Q}). Such *fixed abstraction levels* were used in the CARCASS approach of the previous section. This corresponds to the top half of the diagram, where all \tilde{Q}^i have the same logical structure, and only differ in their values. In both approaches, policies can be generated online, either by induction on the same learning samples, or by deduction from the current value function. Because of the inductive step needed to generate the next Q -function by sampling from the current, essentially all current algorithms are *approximate*. In contrast, a number of model-based approaches in the next chapter focus on *exact* representations of values functions.

In the following we survey algorithms that use fixed logical abstractions (i.e. PIAGET-1 and PIAGET-2), as well as algorithms that generalize dynamically during learning (i.e. PIAGET-3). In addition, we will encounter several ways to provide an (initial) abstraction level automatically (i.e. PIAGET-0). We distinguish several subdivisions in each of these groups of methods, based on algorithmic and representational differences. One aspect that plays an important role in all approaches, is the form and the amount of *bias* provided to the learner. Especially for dynamic generalization, the learning algorithm must be provided with machinery to build representations, e.g. a language and refinement operators. Even though model-free algorithms do not assume prior knowledge about a model of the environment, the first-order setting still requires more KR efforts than related methods in the propositional setting.

5.3.1 Value-Based Learning on Fixed Abstraction Levels

A first class of value-based learning methods side-steps the difficult task of structural induction by fixing parts of the logical abstraction level before learning. This means that the problem of *model selection* is assumed to be solved prior to the actual task of *parameter estimation* (i.e. PIAGET-1 or PIAGET-2). Fixed abstractions have several advantages. First, the learning problem is more stable when compared to situations in which the representation must be learned alongside the parameters. Second, convergence, error bounds, and other properties of the abstraction level are easier to examine when the representation is fixed. Suitable abstraction levels can be (semi-)automatically generated (i.e. PIAGET-0),

for example from sampling the domain, or by employing adequate domain knowledge, if available. In this section we provide an overview of existing techniques that fix the logical abstraction level and focus on learning parameters for such abstractions. In Section 5.5.3 we describe similar parameter-learning methods for policy-based approaches.

There are some connections with work that combines RL algorithms with PROLOG programs. For example, in the context of his work on PRISMs (Sato, 1995), Sato (2001) solves the classical RL problem of finding the shortest path in a state transition graph, by applying RL algorithms to learn parameters for a logic program that encodes the problem. Here, parameters are attached to program clauses, but the aim is to find ground solutions (e.g. optimal paths through the graph). Recently, Saad (2008) describes a related approach based on *hybrid probabilistic logic programs*. Saad shows that MDPs can be encoded in this formalism, and that trajectories in an RL problem correspond to probabilistic answer sets. In addition, it is shown that such RL problems can alternatively be seen as a SAT problem. Another related approach is the *adaptive linear interpreter* (ADLIN) by Asgharbeygi *et al.* (2005), who use a value-driven inference mechanism combined with RL to deal with the challenge of data-driven *inference* under time constraints. ADLIN uses a generalization mechanism that is based on an additive reward function and class-based generalization. The approach deviates from most other techniques in this book, in that it does not focus on the traditional task of action learning.

5.3.1.1 LOCAL STATE ABSTRACTIONS

In the previous section we have described the CARCASS abstraction (van Otterlo, 2003, 2004a). There are two closely related systems that use similar, fixed abstractions over RMDPs and model-free learning algorithms to learn values and policies. Most closely related is the *logical MDP* (LOMDP) approach by Kersting and De Raedt (2003, 2004). Basically, a LOMDP is a logical representation of RMDPs in terms of probabilistic STRIPS (see Chapter 4, and also Chapter 6 where we employ a similar formalism in the model-based context). Policies in the LOMDP context are equivalent to CARCASS policies, except that the language and inference procedure are simpler. Whereas CARCASSs employ a full PROLOG-like language, LOMDP policies use simple queries (without background predicates and negation) and inference is performed by θ -subsumption. LOMDPs itself are simple instances of the FORMs we have defined in Definition 4.5.5. Kersting and De Raedt employ two types of learning algorithms. The *logical Q-learning* (LQ) algorithm learns action values for rules that are basically CARCASS policy rules. The *logical TD(λ)-algorithm* (LTD) uses a TD(λ)-learning rule (see Section 2.6) for estimating a state value function for abstract states, using a given, logical policy to generate learning samples. Whereas LQ can be used to choose between a given set of different policies, LTD is useful to evaluate a *given* policy for a *given* set of abstract states. Experiments on small BLOCKS WORLD abstractions show, among other things, that LTD(0) converges for finite abstraction levels. Based on the same general BLOCKS WORLD abstraction (i.e. the CARCASS \mathcal{C}_{GBW}), experiments using LTD seem to support our results on that the two actions $\text{move}(A, \text{floor})$ and $\text{move}(A, C)$ are indistinguishable; two policies that differ only in these actions, result in the same state values for the first abstract state in \mathcal{C}_{GBW} . The advantage of our PS algorithm (and accompanying analysis) is that it shows that the reason for this phenomenon is that both actions have the same transition probabilities.

A second method is the *relational Q-learning* (RQ) approach by Morales (2003). RQ

is similar to both CARCASS and LOMDPs in that it provides a fixed abstraction over the joint state-action space of an RMDP. However, in contrast to CARCASSs and LOMDPs, Morales completely separates the state and action spaces. In this representation so-called *r-states* are defined by sets of first-order relations that partition the state space of the RMDP into abstract states. Each *r-action* is a general template, defined by a precondition that specifies its applicability in states. The *r-states* and *r-actions* in fact define a new (abstract) state-action space, if the designer takes care of **i**) that the state space defines a strict partition (i.e. each state of the RMDP belongs to exactly one abstract state) and **ii**) that if an abstract action is applicable to a state belonging to some abstract state, it should be applicable in all states belonging to that abstract state. On the minimized, abstract state-action space a relational version of Q -learning can be naturally applied, in a similar fashion as in the CARCASS representation. In fact, the separate state and action spaces in RQ enable all existing convergence proofs for standard model-free RL algorithms to be immediately carried over to this new abstract space. The representation was tested in maze problems, the Taxi domain (Dietterich, 2000a, see also Section 3.8) and CHESS (King-Rook vs. King), and also in a power generation application (Reyes *et al.*, 2003).

All three approaches (CARCASS, LOMDP and RQ) can be used to learn optimal policies in domains where prior knowledge exist to generate an adequate abstraction. Furthermore, they can be employed as part of other learning methods, for example to learn sub-policies in a given hierarchical decomposition of a value function such as in HRL (see Section 3.8). From all three, CARCASSs provide the most general representation, in the sense that both other representations can be embedded in a CARCASS. LOMDP policies and value functions can be represented using CARCASSs where each rule has only one action head, and where no background knowledge is used. RQ structures can be represented as a CARCASS using additional relations that ensure that all rules form a partition of the state space, and that all action preconditions are fulfilled for all abstract states. However, this will blow up the representation. The representation used in RQ requires a different kind of modeling and further analysis is needed to study which one is most suitable for particular problems. One crucial difference is that whereas in CARCASSs actions are defined locally to an abstract state, actions in RQ are defined more globally. Another difference is that CARCASSs have an order imposed on the abstract states, which can be used for easier modeling of sequential phases in a task, as shown in our TIC-TAC-TOE experiments. Both RQ and CARCASS have shown elegant representations in different domains, whereas the LTD approach has only been applied in the BLOCKS WORLD.

In general, all three use abstraction to transform the underlying RMDP into a much smaller abstract MDP, which can then be solved by modifications of traditional RL algorithms. In fact, the abstraction levels are *exact aggregations* (i.e. partitions) of state-action spaces, and therefore these methods are closely related to the work on *averagers* (Gordon, 1995c). Furthermore, all three could benefit from the work on *model minimization* (Givan *et al.*, 2003, see also Section 3.4) for the automatic construction of the abstraction levels. Most of the abstraction levels in this section can be mapped onto abstractions used in other model-free approaches that do support dynamic generalization (see the next section). For example, a Q -tree in Q-RRL (Džeroski *et al.*, 1998) has a similar semantics as e.g. a CARCASS. This means that the evaluation method from CARCASS could be applied to the Q -tree and that parameters for the induced Q -trees might be learned using RQ. Other learning methods from ILP might be used as well, for example one could use specific learn-

ing techniques to learn structural aspects of PRMs for use in class-based value functions. Morales (2004b) proposes such possible techniques to be used in RQ, but in general, much cross-fertilization is possible between the approaches. For example, the recent technique by Wang *et al.* (2008b) uses MLNs (see Section 4.3.3) as a generic representation for RL.

Some initial approaches have been proposed to automate the generation of logical abstractions such as used in CARCASS. Song and Chen (2006) introduce an approach that can be seen as somewhere in between CARCASS and LOMDP on the one hand and RQ on the other. It uses separate sets of abstract states and abstract actions, resembling the RQ approach. But, when applying an action, its variables must be matched with the variables in an abstract state (as in CARCASS and LOMDPs), possibly generating additional, unintended actions. Furthermore, it enforces that all preconditions for all actions are represented in each abstract state, which seems unnatural if a number of different actions is present in the domain. Song and Chen also devise a technique for manually refining the state space (through the employment of negation on specific conditions) though finding the right conditions seems at least as hard as coming up with the initial state space abstraction. To remedy this problem, Song and Chen (2007) extend their approach with an automated state splitting method. Abstract states that display the largest amounts of *self-loops* are selected and split according to a new condition that is found by generalization of ground samples covered by that state (see further Song and Chen, 2008). Related to this, Walker *et al.* (2007) propose to learn abstract state-action pairs from traces in the AMBIL approach (which stands for *Abstract Model Building using ILP*). Given an abstraction level, AMBIL estimates an approximate model (as in CARCASS's PS algorithm) and based on these statistics, it computes *pre-images*¹⁰, i.e. sets of state-action pairs that lead to some abstract state. AMBIL then generalizes these sets through ILP in order to create new abstract state-action pairs. To ensure that the resulting rules do not overlap, AMBIL orders them according to their creation order. Abstract RMDP generation phases are interleaved with value-based learning, and in each iteration a new representation is generated from scratch. AMBIL is closely related to the adaptive resolution methods described in Section 3.4, though no connections are made by the authors. The combination of heuristics used in AMBIL and the fact that experimental results are mainly based on an essentially propositionally encoded problem, leaves open the question on how it would perform in more relational worlds such as the BLOCKS WORLD, TIC-TAC-TOE or CHESS. As we have seen in the experiments with CARCASS, such worlds require sophisticated, crisp abstractions with complex relational patterns. A third approach in automating the generation of abstractions was based on RQ. Morales (2004a) employed *behavioral cloning* to learn *r-actions* from sub-optimal traces, generated by a human expert in a *flight simulator* application. Similar to AMBIL, this approach was only tested in an application that is essentially a propositional problem, consisting of a number of real-valued features with no (relational) interaction between features. However, both methods are potentially powerful for general, relational domains (see also Morales, 2004b, for more extensive discussion on opportunities to learn abstractions in the RQ approach).

¹⁰Note that these pre-images are computed using ground samples. In Chapter 6 we deal extensively with *regression*, a technique that computes such pre-images on the logical abstraction level.

Algorithm 10 Feature construction (Walker *et al.*, 2004; Srinivasan, 1999).

Require: A training set of state-action- Q -value triples of the Q -function to be learned

- 1: **for each** distinct action a create p ensembles using **do**
 - 2: **for each** ensemble create m features using **do**
 - 3: stochastically create a feature f (Srinivasan, 1999) such that
 - 4: i) it covers more than 25% and less than 75% of the samples
 - 5: ii) it differs significantly in truth values from the other features so far
 - 6: add f to the current predictor
 - 7: Build a model using the m features utilizing an appropriate regression algorithm
-

5.3.1.2 DISTRIBUTED, FEATURE-BASED ABSTRACTIONS

In addition to the CARCASS-like representations, where there is a clear correspondence between abstract states and states of the underlying RMDP, there are approaches that use first-order features as their base representation. Features are widely used in propositional VFA techniques (see Section 3.6) and *first-order* features have been described in Chapter 4. Surprisingly, not many model-free approaches in the first-order setting have used such feature-based representations yet. In Section 6.5.2.1 we survey model-based algorithms using feature-based representations, and here we briefly describe some that use them in model-free algorithms. Such representations offer a more *distributed* representation, which can be used to generalize over object classes and used in *additive* reward function decompositions.

The approach by Walker *et al.* (2004) separates the structural induction of the representation from the actual value function estimation. First a set of first-order features – represented by relational conjunctions over state atoms – is induced. These are then used as input for a regression algorithm that estimates Q -value functions per action. The method resembles standard function approximation methods for RL (see Section 3.6), using first-order features and regularized kernel regression for learning the value function. However, instead of Q -learning, Monte Carlo estimates using hand-coded policies are used as training examples. The use of (semi-) random features in value function approximation was already described in the propositional case by Sutton and Whitehead (1993). The use of a first-order language requires a more principled approach to generate¹¹ a set of features that sufficiently covers the state space. One approach for this was described by Srinivasan (1999) and a general outline for the feature construction can be found in Algorithm 10. More specifically, an *ensemble* of feature sets is created, for increased robustness. The method was tested in a keep-away subtask of (simulated) robot soccer in which 3 players have to keep a ball from being captured by 2 defenders. States are described by, for example, `distToBall(P1, Dist, Game, Step)` and `angleBetween(P1, P2, Ang, Game, Step)` and actions such as `hold` and `pass`.

A more recent approach that uses a feature-based representation, is *relational temporal difference* learning (RTD), proposed by Asgharbeygi *et al.* (2006). Technically, it should be classified as a model-based approach (see the next chapter) because it assumes a known action model. However, the action model is only used for selecting actions, based on a

¹¹Note that here, in the absence of a model, features are inductively created using samples generated in the domain itself. In the next chapter we will describe other approaches that use a symbolic domain model to generate features using deductive techniques.

learned state value function. The learning method and representation are more similar to the previously described techniques, which explains why we treat it here. Asgharbeygi *et al.* employ a set of concept definitions (e.g. PROLOG definitions of predicates) that must be supplied by a designer and assign a utility to each of them. The value function is then represented as a linear combination of the utilities for those concepts that can be inferred¹² from the current state, weighted by the number of times a concept can be inferred from a state. For example, in the game of TIC-TAC-TOE a state value could be computed as the number of `fork` positions plus the number of `win` positions. Now the state value function can be learned by a version of $TD(\lambda)$, using the action model to maximize the expected value of the next state for action selection. As we have mentioned in the TIC-TAC-TOE experiments using CARCASS, (after-)state value functions are very useful because of equivalent effects of different actions. Asgharbeygi *et al.* test their approach on TIC-TAC-TOE and on mini-CHESS. The advantage of the distributed, feature-based representation of RTD is that the value function mapping is – in principle – flexible enough to represent the value function of many different policies. The CARCASS and LQ representations can only switch between a limited set of policies, whereas LTD is focused on estimating the value function for a fixed policy. A downside of the representation used in RTD is that learned value functions and policies are less comprehensible.

5.3.2 Value-Based Learning using Dynamic Generalization

Although fixed abstractions are useful in many respects, the general goal of RL in first-order domains is to have algorithms that learn both their behavior as well as their representations from interaction with an environment. This is a difficult problem, since the agent is assumed to have no prior knowledge about either the transition probabilities or the reward function of the underlying RMDP. A common factor that distinguishes the methods described here from those that were surveyed in the previous section, is that they employ structure-learning algorithms as an integral part of the behavior learning process, as in PIAGET-3 learning. Most methods learn a logical abstraction level that can be used for abstracting value functions. The literature displays a vast number of different representations, structural induction algorithms and value learning algorithms. In order to structure our description of methods we focus on two important dimensions. The first is the *representational* dimension, which covers topics such as **i)** the representation of the states and actions, **ii)** the form and amount of *bias* supplied to the learner in the form of a hypothesis language, distance measure, kernel, background knowledge predicates, or search biases, and **iii)** the representation of abstract value functions and policies. For all three aspects, we are mainly interested in how they come about. The second dimension is the *adaptability* dimension, which is about how representations evolve over time. Some systems may gather samples that they use in batch learning algorithms to induce structures from scratch over and over again, while others can adapt small portions of an existing representation during learning. Based on the representational dimension, we distinguish three kinds of approaches. The first covers methods that use representational devices based on first-order logical languages, and generalization is performed by building logical abstractions, mainly using ILP algorithms. The second category uses different means for generalization, for example by defining distances on ground states. The main focus of the third category is

¹²There seem to be similarities with CMAC models for VFA (see Chapter 3).

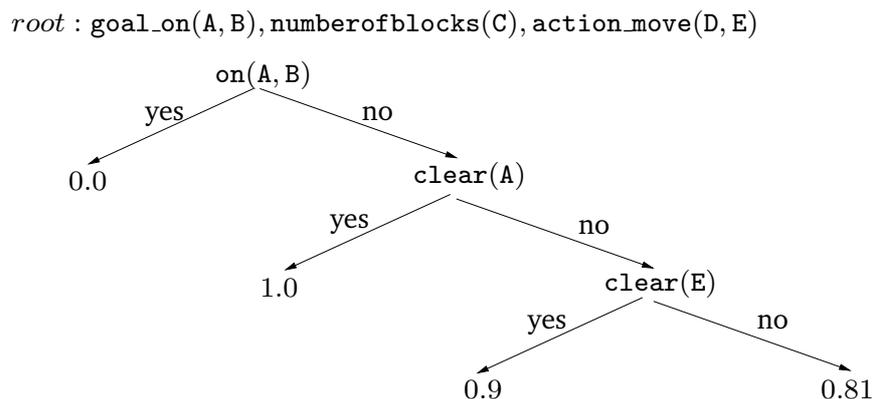


Figure 5.6: A logical Q -tree. The root of the tree contains the current goal for the learner, the size of the world and the action considered. The inner nodes of the tree contain tests in the form of logical atoms that test for different properties of the current state. The leaves of the tree contain the Q -value of the currently considered action in the current state.

on probabilistic methods.

5.3.2.1 LOGICAL ABSTRACTIONS

The Q-RRL¹³ method (Džeroski *et al.*, 1998) is¹⁴ the first approach towards model-free RL in RMDPs. It is a straightforward combination of standard Q -learning and an ILP algorithm for generalization of Q -functions. More specifically, TILDE-RT (Blockeel *et al.*, 1998) is used as a representational tool to store the Q -function in a first-order logic decision tree (see Section 4.2.3.1), which is called a Q -tree. Q-RRL collects experience in the form of state-action pairs with corresponding Q -values. During an episode, actions are taken according to the current policy, based on the current Q -tree. All newly encountered state-action pairs are stored, while the values of already encountered pairs are updated according to the Q -learning algorithm. After each episode a decision tree is induced from the example set. The resulting tree contains nodes that are basically PROLOG queries and nodes in the tree can share variables. The representation language used for the Q -trees employs background knowledge and a declarative bias. Individuals (i.e. constants) are not referred to in the tree itself directly, but only through the variables of the goal. See Figure 5.6 for an example of such a tree. Note also that the goal is part of the root of the tree; in this way tests in leaf nodes have access to variables mentioned in the goal. The tree can be transformed into the following PROLOG program (essentially a decision list):

```

qvalue(0) :- goal_on(A,B), numberofblocks(C), action_move(D,E), on(A,B), !.
qvalue(1.0) :- goal_on(A,B), numberofblocks(C), action_move(D,E), clear(A), !.
qvalue(0.9) :- goal_on(A,B), numberofblocks(C), action_move(D,E), clear(E), !.
qvalue(0.81).

```

Q-RRL was tested on a number of small, deterministic BLOCKS WORLD instances that have now become some of the standard benchmark problems in the field. The declarative bias available to the system contained much domain-specific knowledge, such as about the

¹³The general area of solving RMDPs is now often termed *Relational Reinforcement Learning*, but this first method uses exactly the same name. We will therefore use the acronym Q-RRL for this system.

¹⁴See also (Driessens, 2001a).

amount of and heights of towers, the number of blocks and about which are the blocks on top of the towers.

One problem with Q -functions however, is that they implicitly encode a *distance* to the goal, and they are dependent on the domain size (see Chapter 4 on *families* of RMDPs). A Q -function represents more information than actually needed for selecting an optimal action. For this reason, in subsequent papers an extension to policy induction was developed under the name P -learning (Džeroski *et al.*, 2001a,b; Driessens, 2001b). Based on the current Q -function and a training set, a P -function is computed. For each state s occurring in the training set, all possible actions in that state are evaluated and a P -value is computed as

$$P(s, a) = \begin{cases} 1, & \text{if } a = \arg \max_{a'} Q(s, a') \\ 0, & \text{otherwise} \end{cases}$$

Now, a new tree is learned to *classify* actions as *optimal* (1) or *not optimal* (0). This new P -tree represents the best policy relative to the current Q -tree. In general, it will be less complex and it will generalize better over domains with different numbers of blocks. In fact, independently around that same time, Lecoeuche (2001) showed similar results on the advantages of learning policies for RMDPs in a dialogue system application (see Section 5.5 for more on this approach).

Cole *et al.* (2003) use a similar setup as Q-RRL, but upgrade the representation language to *higher-order logic* (HOL, see Section 4.2.2.3) (Lloyd, 2003). As in Q-RRL, a logical tree representation is induced to represent a Q -function, but in this case the HOL tree learner ALKEMY (see Lloyd, 2003) is used. An advantage of the approach is that a powerful language can be used to specify the domain and provide bias for the learner. A disadvantage is that learning in HOL involves much design and specification (i.e. bias) in the form of a *predicate rewrite system*, for even the simplest applications. Cole *et al.*'s approach uses a combination of both Q -learning and P -learning, but it employs a *different* hypothesis language for the value function and policy, allowing for more flexibility. Furthermore, it considers a more general context in which policies may be available in the form of a *policy library* before learning. The approach was tested on somewhat more complex BLOCKS WORLD problems, for example to correctly place a number of blocks, and furthermore on changing tasks and environments, and was shown to generate comprehensible policies.

Despite some success in learning good policies using the Q-RRL (and its HOL successor), the approach comes with a number of severe shortcomings. First, after each episode, a new Q -tree is induced from the examples, which is clearly not efficient. Second, the set of examples is constantly growing; all state-action pairs have to be memorized. Third, updating values for already experienced state-action pairs requires searching through the whole example set. Fourth, generalization is not done in an efficient way. When updating a value for one state-action pair, the values of all pairs in that leaf should be updated, but in Q-RRL only those pairs are updated that are experienced exactly¹⁵ again.

The *incremental* extension to Q-RRL that solves these problems is the TG-algorithm (Driessens *et al.*, 2001a,b). This algorithm can be seen as a first-order extension of the G -

¹⁵Note that this also entails that updates in Q-RRL do not function as intended with abstraction. When updating the value of a state-action pair covered by a leaf node, one would expect that the Q -value for *all* pairs covered by that node would be updated. But this is not the case, since a pair is only updated if it is experienced again. Generalization only happens when the tree is induced again.

algorithm (Chapman and Kaelbling, 1991, see also Section 3.6.2.3), and it *incrementally* builds the same kind of trees as TILDE-RT. Each leaf in the tree is now augmented with statistics¹⁶ about Q -values and the number of positive matches of examples. A node is split when it has seen enough examples and a test on the node's statistics becomes significant with high confidence. This mechanism removes the need for storing, retrieving and updating individual examples. Generating new tests is much more complex than in the propositional case, because the amount of possible splits is essentially unlimited and the number of possibilities grows further down the tree with the number of variables introduced in earlier nodes. Although a declarative language bias limits the number of possible tests (see also Section 4.3.2 on such bias and refinements), there are still many tests to be considered and tests cannot be undone. Due to incomplete experience at the start of learning, the possibility of introducing tests irrelevant for the final (possibly optimal) policy, is often inevitable. Still, TG is much faster than the original Q-RRL on the same deterministic BLOCKS WORLDS, but a problem is how to set the *minimal sample size* that determines when to split a node. A larger value means a smaller representation, but also slower convergence. In order to cope with the increased computational complexity of TG, *query packs* (see Section 4.2.2) were used.

Some other approaches are direct extensions of the work on TG. First of all, Ramon (2005b,a) studies the convergence properties of tree-based algorithms such as TG, relative to the concept language that is used to build the trees. Although convergence can be proved, one of the remaining challenges is to formalize those circumstances in which learning will find a tree that is more compact than a ground representation. An interesting practical extension was described by Goetschalckx and Driessens (2007) who introduce *costs* for some complex predicates that could be used in inducing the tree. In contrast to standard cost-based extensions in MDPs, Goetschalckx and Driessens' motivation is to mix *planning* and *learning* by providing some complex *tests* to the learner, that involve various n -steps look-ahead properties of states. For example in CONNECT-FOUR (the domain used by the authors), such a complex test could be about "will the learner win in three moves when moving to this position?". Such tests can be used in the TG algorithm to split leaf nodes, and the corresponding Q -value will be lowered by the cost of the test. The work is still preliminary, but it is potentially¹⁷ useful in trading-off¹⁸ the knowledge used in learning and the computational complexity of that knowledge.

A last extension of TG that we consider here tackles the problem of the *irreversibility* of the splits that are added to the tree while learning. Ramon *et al.* (2007) extend the TG algorithm with *tree restructuring* operations based on ideas of the *incremental tree induction* (ITI) algorithm by Utgoff (1997) and concepts from *theory revision* in ILP. New operations in this new algorithm TGR include the pruning of a leaf or a whole subtree, and the revision of an internal node. In order to carry out these operations, statistics are now stored in all nodes in the tree (in contrast to TG that only stores these for the

¹⁶Each node contains the number of examples on which each test succeeds, the sum of their Q -values, and the sum of their squared Q -values. In addition, the general Q -value of the node itself is stored.

¹⁷Although not the same (and not discussed by the authors), the approach has some similarities with *hierarchical* approaches in RL. The complex tests bring a certain flavor of useful sub-policies of which the results (i.e. the costs of executing) are used without actually executing them.

¹⁸The idea of this approach was already present in the policy gradient method by Itoh and Nakamura (2004) who attached costs to certain memory altering actions such that the algorithm has to learn when to use such planning-ahead actions.

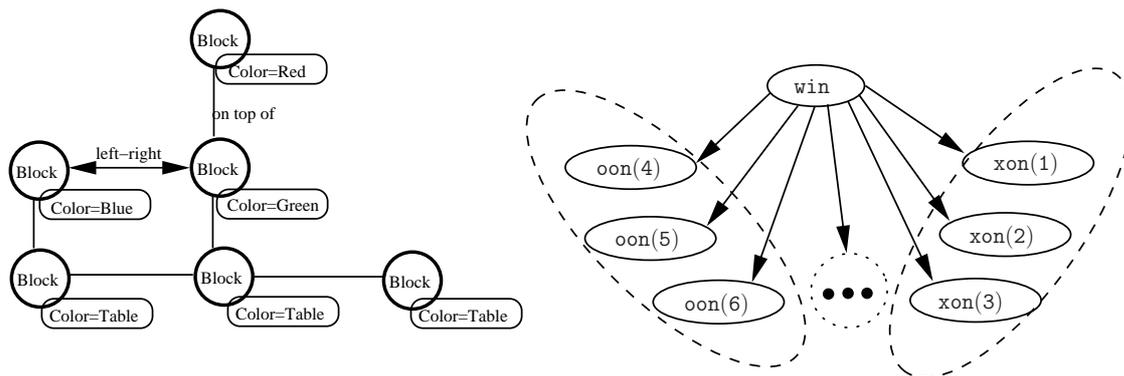


Figure 5.7: **a)** An example graph representation of a state used by Dabney and McGovern (2007). Note that no use is made of different constants for blocks. The state representation bears some similarities with the example we used for the robot FREGE in Chapter 1. **b)** A (part of) a relational Bayesian network representation of the value function by Sanner (2006a). It shows the features in the game of TIC-TAC-TOE. The dashed line shows two different sets of node joins that can be useful in the game, to detect winning lines for both players.

leaf nodes). Special care is to be taken with variables in the tree when building and restructuring the tree. Experiments using the BLOCKS WORLD and TIC-TAC-TOE show that TGR reacts somewhat faster than TG when the task is changed while learning, but more (experimental) analysis is needed to study how the methods compare in terms of convergence speed, tree size and computational complexity.

A recent method that has used restructuring operations from the start is the *relational* UTREE (RUTREE) algorithm by Dabney and McGovern (2006, 2007). RUTREE is a first-order extension of the UTREE algorithm by McCallum (1995, and see also Section 3.6.2.3) and (as is TGR) on the ideas of ITI. The state language used by RUTREE is *graph-based*, and more specifically using *attribute* graphs. An example of a state representation is depicted in Figure 5.7a. The learning algorithm follows the outline of the propositional UTREE algorithm, now adapted to the graph-based state representation. Because RUTREE is instance-based, tests can be regenerated when needed for a split such that statistics do not have to be kept for all nodes, as in TG. Another interesting aspect of RUTREE is that it uses *stochastic sampling* (similar to the approach by Walker *et al.* (2004), see also Algorithm 10) to cope with the large number of possible tests when splitting a node. Combining these last two aspects shows an interesting distinction with TG (and TGR). Whereas TG must keep all statistics and consider all tests, RUTREE does not keep all statistics and considers only a limited, sampled set of possible tests. In return, RUTREE must often recompute statistics which can be computationally expensive. Experiments were conducted on simple BLOCKS WORLDS and on the partially observable BLOCKS WORLD used by Finney *et al.* (2002b,a) (see also Section 4.1.3.3). Results on the latter domain show that the tree-restructuring operations, as well as the instance-based representation, enable RUTREE to outperform previous approaches. Furthermore, it was tested on TSUME-GO.

One additional algorithm that is based on logical abstractions is the *first-order* XCS (FOXCS) system by Mellor (2005b,a, 2007, 2008), which is a *learning classifier system* (e.g. see Lanzi, 2002) using definite clauses as a representation language. Each clause $A \leftarrow S$ consists of a conjunction of atoms that describe an abstract state S and an action atom A . In this sense they are similar to the rules used in LOMDP policies (see Sec-

tion 5.3.1) but with a different semantics. Each rule is augmented with an *accuracy* and each time an action is required, *all* rules that cover the current state-action pair considered, are taken into account. Based on the accuracies of the rules, a rule is selected and an action is applied. Structural adaptation of the rules during learning is done by an *evolutionary learning* component that can modify rules in a similar way as ILP algorithms, using refinement operations and based on language and search biases (evolutionary algorithms are explained in Section 5.4). Experimental results on BLOCKS WORLD instances show that FOXCS can compete with other approaches such as TG.

5.3.2.2 DISTANCES AND KERNELS

Instead of building logical abstractions based on a given hypothesis language, several methods use other means for generalization over states modeled as first-order interpretations. They are similar to the *intermediate* approaches in Section 4.1.3.3 in the sense that they move to another space to define new ways of generalization. However, in contrast, they do make use of the relational nature of the states, albeit in a different way.

The *relational instance based regression* method (RIB) by Driessens and Ramon (2003) uses *instance-based learning* (Aha *et al.*, 1991) on ground states. The Q -function is represented by a set of well-chosen experienced examples. To look-up the value of a newly encountered state-action pair, a *distance* is computed between this pair and the stored pairs, and the Q -value of the new pair is computed as an average of the Q -values of pairs that it resembles. Special care is needed to maintain the right set of examples, by throwing away, updating and adding examples. Instance-based regression for Q -learning has been employed for propositional representations before but the challenge in first-order domains is defining a suitable *distance* between two interpretations. For RIB, a *domain-specific* distance has to be defined beforehand. For example, in BLOCKS WORLD problems, the distance between two states is computed by first renaming variables, by comparing the stacks of blocks in the state and finally by the *edit distance* (e.g. how many actions are needed to get from one state to another). Other background knowledge or declarative bias is not used, as the representation consists solely of ground states and actions. RIB was tested on BLOCKS WORLDS in a similar fashion as TG. García-Durán *et al.* (2008) used a more general instance-based approach in a policy-based algorithm (see Section 5.5). Katz *et al.* (2008) describe a very interesting application of instance based, relational RL in a robotic manipulation task. They employ a relational Q -function based on a similarity measure that is defined in terms of isomorphic subgraphs induced by the relational representation.

Later, the methods TG and RIB were combined by Driessens and Džeroski (2005), making use of the strong points of both methods. TG builds an explicit, structural model of the value function, which is dependent on the language bias and – in practice – can only build up a coarse approximation. The RIB method is not dependent on a language bias and the instance-based nature is better suited for regression. However, it does suffer from large numbers of examples that have to be processed. The combined algorithm – TRENDI (which stands for TREes AND Instances) – builds up a tree like TG but uses an instance-based representation in the leaves of the tree. Because of this, new splitting criteria are needed. Because both the language bias for TG and RIB are needed for TRENDI, more things have to be specified. However, on deterministic BLOCKS WORLD examples, it is shown that the new algorithm performs better on some aspects (such as computation time) than its parent techniques. Note that the RUTREE algorithm is also a combination

of an instance-based representation combined with a logical abstraction level in the form of a tree. No comparison has been reported on yet. Rodrigues *et al.* (2008) recently investigated the online behavior of the RIB algorithm and their results indicate that the number of prototypes in the system is decreased when per-sample updates are used.

Compared to RIB, Gärtner *et al.* (2003) take a more principled approach in the KBR algorithm (which stands for *kernel-based regression*) to distances between relational states and use *graph kernels* (see Gärtner, 2003) and *Gaussian processes* for value function approximation in relational RL. Whereas Walker *et al.* (2004) use kernel regression over the set of induced first-order features, KBR defines a kernel on the actual states and actions. A simple view on kernels is that they define a distance between instances, which are relational interpretations representing states and actions in this approach. Each state-action pair is represented as a *graph* (see Figure 4.5 in Chapter 4) and a product kernel is defined for this class of graphs. The kernel is wrapped into a Gaussian radial basis function, which can be tuned to regulate the amount of generalization. Ramon and Driessens (2004) examined the numerical stability of this approach. KBR was tested on BLOCKS WORLD problems in a similar fashion as Q-RRL. Later it was also used for the game of TETRIS, though there the problem was modeled using attribute-value vectors and no use of the relational aspects of the problem was made (Driessens *et al.*, 2006b).

Although KBR, RIB and TRENDI are interesting from many points of view because they offer new, and other, ways of generalizing experience, they lose the comprehensibility of the logical representations such as used by TG and FOXCS. In addition, because of the lack of variables in RIB and KBR (and TRENDI to some extent), it is no longer possible to directly learn a generalized policy that can be transferred to domains of different size.

5.3.2.3 PROBABILISTIC APPROACHES

Three additional, structurally adaptive, approaches are SVRRL by Sanner (2005), QLARC by Croonenborghs *et al.* (2004), and MARLIE by Croonenborghs *et al.* (2007b). All three learn and use probabilistic information about the environment to optimize behavior, yet for different purposes.

SVRRL is concerned with model-free learning in undiscounted, finite-horizon domains in which there is a single terminal reward for failure or success. These assumptions enable viewing the value function as a *probability of success*, such that it can be represented as a *relational naive Bayes network*. The structure and the parameters of this network are learned simultaneously. The parameters can be computed using standard techniques based on the maximum likelihood criterion. Two structure learning approaches are described for SVRRL and in both relational features (ground relational atoms) can be combined into joint features if they are more informative than the independent features' estimates (see Figure 5.7b for an example), under an MDL measure. The assumption on the independence of features might be strong for some domains, although it makes the system very efficient and robust. SVRRL was tested on BACKGAMMON and converged to a competitive level of play using a small number of training games. An extended version of SVRRL uses ideas from a well-known *datamining* (DM) technique called APRIORI to focus structure learning on only those parts of the state space that are frequently visited. The new algorithm, DM-SVRRL (Sanner, 2006a) incorporates a search for frequently (positively) co-occurring features based on frequency counts. Whenever a pair is frequent (according to a minimum threshold) a joint feature is built and the algorithm keeps track of its joint

frequency (which can later be used to build even larger features). DM-SVRRL was tested extensively on three games, which are BACKGAMMON, OTHELLO and TIC-TAC-TOE. Results show that online feature extraction is beneficial, that the APRIORI-style feature extraction outperforms near-random feature extraction in large problems, and that the learned value function is comprehensible due to the simple structure of the Bayesian network.

SVRRL does not use TD-like learning to estimate a value function, but instead employs probabilistic reasoning based on a restricted representation of that value function. In contrast, a related approach that uses probabilistic information to aid in a Q -learning algorithm is the QLARC approach by Croonenborghs *et al.* (2004) (which stands for *Q-Learning Agent with Reasoning Capabilities*). The authors discuss *informed* RL in which the learner tries to use as much knowledge as can be gathered in the environment, in this case by learning features that predict a probability of achieving a reward or the truth of other features in the next state. First, features close to the goal area are learned, and subsequently features that are correlated with already discovered features. The dependencies between features are represented by probabilistic rules, for example $\text{prev_state}(R, S), \text{move}(R, S, A, B) \rightarrow \text{on}(S, A, B)$ denoting that doing action $\text{move}(A, B)$ in state R makes it likely that in the next state $\text{on}(A, B)$ holds. An approximation of the expected reward can be obtained from a look-ahead using the rules. Initial experiments in the BLOCKS WORLD show that the *partial model* (as opposed to a full transition model) increases performance compared to the model-free RIB method. A more rigorous extension of QLARC is the MARLIE (which stands for *Model-Assisted Reinforcement Learning in Expressive Languages*) method¹⁹. Here, the goal is the same, but the authors take a more general approach to representing and learning the model. For each predicate in the language of the learning system, a *probability tree* is learned in an incremental way, using a modified version of the TG algorithm. Such a probability tree represents for each ground instance of a predicate the probability that it will be true in the next state, given the current state and action. Using the model amounts to do a look ahead some steps in the future using an existing technique called *sparse sampling*. The original TG algorithm is used to store the Q -value function. Experiments in both the BLOCKS WORLD and a logistics domain similar to the one we use in Chapter 6, show the viability of using partial models and also that using lookahead is beneficial even when the model is less accurate.

5.3.3 Discussion of Model-Free, Value-Based Techniques

Since the initial work on Q -RRL, many new representations and algorithms have been reported in the literature²⁰. An interesting development is the incorporation of structural *adaptation* of logical structures while learning. Obviously, RL problems come naturally with some amount of *concept drift* because the initial policy and value structure will be different from the final (optimal) structures. Algorithms such as TG are able to cope with some of the problems associated with changes in the distribution of learning examples, but in the end they do this by a possibly unbounded growth of the structures that represent the value function. Algorithms such as TGR and RUTREE are able to *restructure* their representation over the course of learning, which provides a huge advantage in relational

¹⁹This approach is very similar to the work by Gardiol (2003) which uses sparse sampling on learned transition rules in a rule-based setting.

²⁰The work on RRL, TG, RIB and KBR has also been described in the thesis by Driessens (2004). See (van Otterlo, 2004b; Driessens, 2005) for reviews of this work.

RL, and especially when the dynamics of the environment may change over time.

An important aspect in all approaches is the representation used for examples, and how examples are generalized into abstractions. Those based on logical abstractions have a number of significant advantages. The most prominent one is that they can generalize over domains of different size, through the use of variables in the abstractions and an additional P -learning type learning step. On the other hand, logical abstractions are less suitable for finegrained regression. Methods such as TG have severe difficulties with very simple BLOCKS WORLD problems such as learning a policy for the $\text{on}(A, B)$ -task. Highly relational problems such as the BLOCKS WORLD require complex patterns in their value functions (see also the next chapter for examples) and learning these in a typical RL learning process is difficult. Other methods that base their estimates (in part) on instance-based representations, kernels or first-order features are more suitable because they can provide more smooth approximations of the value function, yet are less comprehensible and it is less easy to transfer their learned knowledge (e.g. a policy) to other problems. Hybrid approaches such as TRENDI or RUTREE or probabilistic approaches such as SVRRL may provide a general solution. However, no matter which type of representation or generalization method is used, all approaches have to specify some kind of bias, in the form of a hypothesis language, background knowledge predicates, distance measure or kernels. A detailed analysis of the (representational) requirements of algorithms relative to their performance would be very interesting and useful. Furthermore, there are many ideas lying around in propositional adaptive resolution and model-minimization approaches (see Section 3.4) that could be used in the relational RL context. Additionally, the use of datamining techniques to generate useful features from data (e.g. as in DM-SVRRL) could prove useful in general.

A more general comparison on the methods – both in terms of convergence speed and computational complexity – would be very useful, but at the moment it is too early to draw any conclusion (but see Mellor, 2007, for some comparison of methods), which is why we have confined ourself to a description of the representational and algorithmic aspects of the approaches. For example, most approaches use BLOCKS WORLDS as a test domain, but the language and available bias differs much from approach to approach. Even for a domain such as TIC-TAC-TOE, which is used by CARCASS, RTD, SVRRL and TGR, the differences in modeling, the opponent, the language bias and so on, are too diverse to make any good comparison. Both the general RL community, as well as the first-order (probabilistic) planning community have come up with benchmark problems and standardized problem descriptions in order to compare approaches and it would be highly relevant for the relational RL community to focus on such standardized domains.

An additional remark on representation concerns some of the promises of relational RL: *comprehensibility* and *transfer* of learned structures to new domains. A number of methods (such as Q-RRL, TG, CARCASS) chooses an explicit, logical representation of learned structures which can be examined and possibly transferred to other domains. Methods using kernels, where the featural abstraction is wrapped into a kernel regression engine, however, are less suitable for this. This does not have to be considered to be a disadvantage, because they aim at using powerful approximation engines. Nevertheless, the amount of comprehensibility or generality of learned structures will influence the possibility of understanding the acquired knowledge or even reusing them in other domains, or as subtask of larger tasks (such as in hierarchical RL and general agent architectures).

Although the HOL approach generates very comprehensible policies, a question remains whether such a powerful language – which is harder to manipulate – is needed for most relational RL applications.

5.4. GREY: Evolutionary Policy Search in Relational Domains

In the previous sections we have explored value-based approaches to learn policies for RMDPs. In the current section we describe an algorithm that searches for policies directly in *policy space*. Policy-based approaches are useful for several reasons. A most prominent reason is that policies are often much smaller than value functions, and that they often generalize over a complete RMDP family. Experiments with various types of *P*-learning mechanisms (see Section 5.3.2) support this. Especially in first-order domains, value functions are hard to represent exactly (e.g. see Kersting *et al.*, 2004, and also Chapter 6). Take for example the goal of clearing block a in a colored BLOCKS WORLD. A value function has to represent all the colors of the blocks in the stack above a. A policy merely has to represent whether there is something above a. This can result in a huge compression and makes the search space for a policy-based algorithm much smaller than that of a value-based approach. A second feature of policy-based approaches is that they are less affected by the properties of the environment (e.g. such as Markovian constraints). Learning is only focused on obtaining a structure that computes suitable actions for each state, and not on approximating all values or transition probabilities of the underlying RMDP.

There are several ways to learn policies directly (see Section 3.7) and here we explore the idea of using an *evolutionary algorithm for reinforcement learning* (EARL) algorithm, called GREY (Muller and van Otterlo, 2005; Muller, 2005). We use an approach that searches in the space of first-order decision list policy representations using *genetic algorithms*. The quality of the policies is tested by sampling the underlying RMDP, so no explicit state values need to be computed nor stored.

5.4.1 Evolutionary Search and ILP

Evolutionary computation (EC) is a population-based, stochastic, iterative optimization technique based on principles of biological evolution such as 'survival of the fittest', genetic inheritance, phenotypes and so on. In computer science, it is often used to solve hard, combinatorial optimization problems, with examples in *scheduling*, *time-tabling*, *planning* and *classification*. Since the work of Holland (1975), a huge amount of techniques has been developed with many specialized sub-fields (see Goldberg, 1989; Mitchell, 1996; Bäck, 1996; Nolfi and Floreano, 2000, for some pointers to the literature).

Out of the many different types of algorithms, *genetic algorithms* (GA) and *genetic programming* (GP) are most common, and both have been used for typical planning domains including RMDPs (see later). Although they differ in the representational aspects, the general structure of both types follows the outline of Algorithm 11. To explain the algorithm, let us use the generation of neural networks for a classification task as an example (see also Section 3.6.2.2). Each *individual* is a neural network. Usually some dozens of individuals form a *population*, though some algorithms use considerably larger populations. The neural network itself is the *phenotype* of the individual. Genetic algorithms operate on a general *genotype*, which is usually a bit-string representation (i.e. a *chromosome*) of the individual. Each individual in the population is evaluated on our given classification task.

Algorithm 11 General scheme for GA (and GP).

- 1: **initialize** a population of individuals
- 2: **evaluate** all individuals in the population
- 3: **repeat**
- 4: select fit individuals as **parents**
- 5: apply **recombination** on pairs of parents
- 6: apply **mutation** on selected individuals
- 7: **evaluate** offspring
- 8: insert offspring into the population
- 9: **until** some predefined stopping criterium
- 10: **extract** a solution from the population

In this case, predictive accuracy would be a good measure on how well the neural network performs. In general, an estimate of how good an individual is, is given by a (domain-dependent) *fitness function*. Now, based on the fitness estimates of individuals, *parents* are selected to create *offspring* for the next generation. The recombination of two parent individuals is performed by exchanging certain parts of the individuals. This is done by cutting the chromosomes at random points and exchanging the part before the cut between parents. For example, when individuals are neural networks, recombination essentially swaps substructures of the parent neural networks, thereby generating two networks that inherit structural aspects of both parents. A *mutation* operator then modifies some small parts of individuals by swapping bits randomly in the chromosome (e.g. corresponding to a modification of a connection between two randomly selected neurons) to introduce some stochasticity in the search, to explore new possibilities. Recombination and mutation generate a new population and this process is iterated until the performance of the best individual of the current population is sufficient for the task. It has been shown that such evolutionary algorithms can result in the increased propagation of good *building blocks* (i.e. substructures that appear in individuals with high fitness) over the generations. A general distinction between approaches is that between *Pittsburgh* approaches, where individuals represent complete solutions (e.g. neural networks), and *Michigan* approaches, where individuals represent only a part of the solution (e.g. individual neurons). In Section 3.6.2.2 we have seen both types of approaches, and an example using both types in one algorithm is the SANE algorithm (Moriarty and Miikkulainen, 1996). Note that Michigan approaches require an additional step to assemble the final solution from the individuals in the population.

In Section 3.7 we have described several approaches that use evolution to find good policies in RL problems, where an intuitive fitness measure is defined in terms of rewards. All the advantages carry over to first-order domains, such as the ability to deal with non-Markovian problems, the avoidance of bootstrapping and an easier credit assignment specification (e.g. see Schmidhuber, 2000; Moriarty *et al.*, 1999). As explained earlier, individuals in the first-order setting are usually represented as a decision list, essentially a PROLOG program. Now the interesting fact is that because EC is a search-based technique, there are many similarities with the ILP algorithms in Section 4.3.2 and we can make use of that in developing EC algorithms for policy search in RMDPs. ILP uses essentially a deterministic search process to find PROLOG programs, by *refining* parts of such programs, guided by the definition of a hypothesis language, search and language biases and the properties

of a *lattice* on the space of clauses. EC uses a similar search for PROLOG programs, though it keeps a population of solutions, and the search is highly randomized. The structural adaptations made by both algorithms are very similar, though decisions on how and when to apply refinement operators differ very much between the two. Divina (2004, 2006) describes the relations between EC and ILP in full detail and surveys several algorithms that solve ILP problems through EC. In the following, we report on our findings on using evolutionary search for policies in RMDPs.

5.4.2 GREY's Anatomy

GREY is a straightforward combination of a standard GA, first-order decision list policy representations, and ILP refinement operators. The general outline of applying GREY to (a family of) RMDPs follows closely Algorithm 11. Each evaluation step computes the fitness of each individual in the population by applying its policy in the RMDP. To get reliable estimates of this fitness, we use a training set that consists of a number of instantiations of the RMDP family. A training environment $\tau = \langle \mathcal{C}, s_0, t_{max} \rangle$ consists of a set of objects \mathcal{C} , an initial state s_0 and a maximum number of timesteps t_{max} . The objects \mathcal{C} correspond to the constants in the definition of an RMDP (e.g. the blocks in a BLOCKS WORLD). A training set consists of a number of training environments. A goal \mathcal{G} is specified for all environments simultaneously. The initial state s_0 defines the starting state of the environment; from here every policy is allowed a maximum number of timesteps t_{max} to reach a goal state. An initial state cannot be a goal state, so $s_0 \notin \mathcal{G}$. In the following, we first describe all the standard GA components of GREY, now adapted to first-order representations of policies, and we continue with an experimental evaluation.

5.4.2.1 INDIVIDUALS: POLICIES AND CHROMOSOMES

Each individual in the population corresponds to a policy, modeled in a similar way as a CARCASS policy (see Definition 5.2.5), i.e. a list of rules of the form

$$conditions \rightarrow action$$

where the condition part consists of a (possibly empty) conjunction of literals. The arguments of the action may be constants or variables, but all variables must occur in the condition too. A rule proposes one or more ground actions if it fires, i.e. if a substitution θ for the condition part exists, matching the state of the environment. If the condition part is empty, the rule always fires. Duplicates are removed from the set of proposed actions, but illegal actions (not valid because of the preconditions of the action) are accepted – if executed, the environment does not change. All policy rules together propose a set of sets of actions in this way: the *proposed actions set*. Since more rules in a policy may propose the same ground action, this set may contain a ground action multiple times.

EXAMPLE 5.4.1 ► Consider the following policy for the BLOCKS WORLD:

$$\begin{array}{ll} \text{clear}(a), \text{clear}(b) & \rightarrow \text{move}(a, b) \\ \text{block}(X) & \rightarrow \text{move}(X, \text{floor}) \\ & \rightarrow \text{move}(d, \text{floor}) \end{array}$$

Let s be a state in which blocks b , d and a form a tower, and block c is on the floor. The first rule does not fire in s , since block a is not clear, and therefore no ground actions

are produced by this rule. The second rule fires four times, since four substitutions for the condition part exist: $\theta_1 = \{X/a\}$, $\theta_2 = \{X/b\}$, $\theta_3 = \{X/c\}$ and $\theta_4 = \{X/d\}$. The set of ground actions $\{\text{move}(a, \text{floor}), \text{move}(b, \text{floor}), \text{move}(c, \text{floor}), \text{move}(d, \text{floor})\}$ is produced as a result; the first and fourth action are illegal. The third rule always fires, because of the empty condition part, so this rule produces a set of ground actions with one element: $\{\text{move}(d, \text{floor})\}$. Since the definition of the move action requires block d to be clear, it is an illegal action. Summarizing, the proposed actions set of the policy is: $\{\emptyset, \{\text{move}(a, \text{floor}), \text{move}(b, \text{floor}), \text{move}(c, \text{floor}), \text{move}(d, \text{floor})\}, \{\text{move}(d, \text{floor})\}\}$.

Now, for picking an action from the proposed actions set, there are two alternatives. The first is *probabilistic action selection* in which the action is selected randomly from the proposed actions set. A ground action proposed by more rules has higher probability of being selected: in the example, action $\text{move}(d, \text{floor})$ has probability $\frac{2}{5}$ of being selected, while the other actions have probability $\frac{1}{5}$. A second choice is *deterministic action selection*, where the semantics of a policy is equivalent to that of a decision list, as in CARCASS. This means that in the example, the actions proposed by the second rule each have probability $\frac{1}{4}$ of being selected for execution. Deterministic action selection has the advantage of always having the same rule proposing the executed action in identical situations (i.e. the same state of the environment). Additionally, the method is less computational intensive than probabilistic action selection, since it is not necessary to calculate the set of actions for each rule in the policy. The drawback is that rules near the end of the decision list will never get the chance to prove their quality. Evolution in these rules is less effective, reducing the rate of exploration in comparison to probabilistic action selection. In the experiments we compare both approaches.

If no rule matches the current state of the system and all sets are empty, a *seeding operator* is executed, creating a new rule for the policy. This operator creates a rule with an empty condition part and a ground action, i.e. of the form

$$\text{true} \rightarrow \text{action}(\text{obj}_1, \dots, \text{obj}_n)$$

The action is chosen randomly from the set of all possible ground actions in the current state of the environment. The empty condition part implies that the rule does fire in the current state. A pruning operator is used to remove unused rules. If a rule has not fired for a number of episodes, it is deleted from the policy. As a result, this operator removes rules which have a logical expression in the condition part which is always false. Additionally, in case of deterministic action selection, rules that have not had the chance of firing are removed, reducing the length of the policy.

Every policy is mapped directly onto a chromosome by letting every gene represent one rule. Since a policy may contain a non-limited number of rules, a chromosome does not have a fixed length. For example, a policy consisting of at least six rules can be encoded as follows:

r_1	r_2	r_3	r_4	r_5	r_6	\dots
-------	-------	-------	-------	-------	-------	---------

5.4.2.2 EVALUATION OF INDIVIDUALS: FITNESS FUNCTION

Our fitness function for a policy π has three components. The first measures the amount of reward collected during an episode (r_π). The second measures how quickly it reaches

a goal state, i.e. the number of steps taken (t_π). The third aspect incorporates an MDL measure, i.e. the size of the policy structure in terms of the number of rules ($|\pi|$). The size of the policy should only become important when choosing between two individuals with roughly the same properties in the first two aspects.

First, the goal-reaching performance aspect is calculated. Let $t_{totalmax}$ be the sum of all t_{max} allowed steps for all problems in the training set. A training set may contain any number of training environments and the maximum number of timesteps allowed may differ per training environment, so in this way the relative goal-reaching performance does not depend on these factors. The rewards are assumed to be positive. The goal-reaching of policy π can now be calculated as

$$\text{goalperf}_\pi = \left(1 - \frac{t_\pi}{t_{totalmax}}\right) r_\pi \quad (5.3)$$

For minimum goal-reaching performance, the set of policies having goal-reaching performance greater than 0 are considered: $\Pi' = \{\pi \in \Pi \mid \text{goalperf}_\pi \neq 0\}$. These policies have $t_\pi < t_{totalmax}$, which intuitively means that they reach the goal state in a training environment at least once. Now the minimum goal-reaching performance goalperf_{min} is defined as the smallest non-zero goal-reaching performance (if one exist, otherwise it is 0). The maximum goal-reaching performance goalperf_{max} is simply the highest value obtained by some policy in Π . Now, the relative goal-reaching performance fitness function of a policy π (goalperfrel_π) is calculated. It is 0 if the goal-reaching performance is 0, otherwise it is mapped onto the interval $[0, \dots, \text{goalperf}_{max} - \text{goalperf}_{min}]$. Only, if $\text{goalperf}_{min} = \text{goalperf}_{max}$, the policies that have gained reward need to be differentiated from the policies that have not, otherwise both will have a relative goal-reaching performance equal to 0 (done by assigning those policies a small value ϵ). By subtracting the minimal goal-reaching performance from every policy, the differences between policies are enlarged. Additionally, it determines the influence of the policy size on the fitness of the policy: the better the performance, the more important the policy size. This importance is denoted by the size weight fitness function. In adjusting the fitness to the policy size, first the policy size is related to the largest occurring size in the generation:

$$\text{size}_{max} = |\pi| \mid \forall \pi' \in \Pi : |\pi| \geq |\pi'| \quad \text{and} \quad \text{relsize}_\pi = \frac{|\pi|}{\text{size}_{max}}$$

Then the size weight is calculated:

$$\text{sizeweight}_\pi = 0.2 \frac{\text{goalperfrel}_\pi}{\text{goalperf}_{max} - \text{goalperf}_{min}} \quad (5.4)$$

Here, sizeweight is scaled on an interval from 0% to 20%. These numbers are chosen arbitrarily; finding the best parameters needs testing. The fitness itself is now the relative goal-reaching performance reduced by the weighted relative size:

$$\text{fitness}_\pi = \text{goalperfrel}_\pi (1 - \text{sizeweight}_\pi \text{relsize}_\pi) \quad (5.5)$$

5.4.2.3 SELECTION OF INDIVIDUALS

Every generation consists of n individuals, which are actually n_{mut} mutants and n_{chi} children created out of the previous generation. Every mutant is obtained by mutation of one

parent and two children are obtained by crossover on two parents. This means a total number of $n_{mut} + n_{chi} = n$ parents need to be selected every episode. Parents are chosen from the population by means of *roulette wheel selection*, i.e. where the probability of selecting an individual is proportional to its fitness value. Policies can be used as parents multiple times. Individuals with fitness 0 have probability 0 of being picked, if other individuals with fitness > 0 are available. If only individuals with fitness 0 exist, all parents are chosen randomly from the entire population. Every generation the complete population is replaced by a new population.

5.4.2.4 REPRODUCTION: RECOMBINATION AND MUTATION

The main goal of the reproduction operators is to search in the policy space for the best policies. Both crossover and mutation are used for reproduction. For the first two parents are needed; for the latter one parent suffices.

Recombination. Recombination or *single-point crossover* is applied in GREY by exchanging a number of rules between two policies. Since a chromosome does not have a fixed length, the crossover point may be located between any two rules, before the first rule or after the last rule, dividing the policy into two parts. In the latter two possibilities, one ‘part’ is empty. Recombination then exchanges the first parts between policies. For example, let π_1 and π_2 be the following two policy chromosomes

$$| r_1^1 | r_2^1 | r_3^1 | r_4^1 || r_5^1 | \qquad || r_1^2 | r_2^2 |$$

where the crossover points have been marked by a double line. Swapping first parts results in the following children:

$$|| r_5^1 | \qquad | r_1^1 | r_2^1 | r_3^1 | r_4^1 || r_1^2 | r_2^2 |$$

Recombination maintains the original rule order in the swapped parts, which is particularly relevant for deterministic policies. Crossover is applied to two parents with probability p_c – if no recombination is applied, both parents are copied to the next generation.

Mutation. The objective of mutation is to create new rules and thereby explore new areas in the space of policies. Since the number of different values for a gene is much larger than in a binary string representation, mutation is more important in a relational domain and should occur more often. A first type of mutation operates at the level of policies. A new rule $\text{true} \rightarrow \text{action}(\text{obj}_1, \dots, \text{obj}_n)$ is created where the action arguments are taken randomly from a set of constants in the domain, and it is inserted at a random position in the policy. Note that rules can also be generated by the seeding operator, but both enforce that rules are started with an empty body.

A second type of mutation operates on the genes itself, i.e. the structure of an individual rule. These mutations correspond to some basic ILP refinement operators. Here we use the following five: **i)** add an atom to the body of the rule with new variables, **ii)** unify two variables in the rule, **iii)** turn a body variable into a constant, **iv)** turn an action constant into a variable of the body, and **v)** negate an atom in the body. The mutation operators either specialize the body or generalize the action of a rule; only the negate mutator works both ways. Mutation is applied on a parent with probability p_m per mutation. In principle, a parent can be mutated by any combination of mutators that have their preconditions met. If no mutation is applied, the parent is not changed and as a consequence copied to the next generation.

5.4.3 Experimental Evaluation

There are many aspects that define a test, such as the problem the system has to solve, the parameters for the evolutionary algorithm and the amount of time allowed to the system to evolve. In this section, these aspects are explored and test values are given. The *main test set* is used to search for the best parameter settings for GREY. It includes variants for action selection and different settings for population size, reproduction probabilities and the number of episodes per test. All tests in this set are executed on a single task in the BLOCKS WORLD domain. Next, an implementation of GREY with appropriate parameter settings deduced from the main test set is used to research the influence on the performance of alternatives in fitness function, policy preservation and background bias. The tests are presented as the *alternatives test set*.

BLOCKS WORLD Settings. We use a standard BLOCKS WORLD, i.e. using the predicates `on/2` and `clear/1` to describe the states, and in addition, we can supply GREY with background knowledge about `ontop(X, Y)` (expressing that block X is the top block of the stack containing block Y), `above(X, Y)` (expressing that X is in the same stack as Y , only higher up), `level(X, L)` (expressing that X has $L - 1$ blocks underneath it, where `level(floor, 0)`), `stacksize(X, S)` (expressing that X 's stack contains S blocks), and `talleststack(X)` (expressing that X is in the tallest stack in the state). We consider the `stack`, `unstack` and `on(a, b)` tasks, and a reward of 100 is obtained when reaching a goal state. The training set consists of BLOCKS WORLDS of different size.

Main Test Set. The main test set is executed on the `on(a, b)` task in the BLOCKS WORLD. For n blocks, the maximum number of steps needed by an optimal policy to execute the task is n for all initial states. The maximum number of steps given to GREY to solve the problem is $t_{max} = 2n$. The predicates that can be used in the body of the rules are `block/1`, `on/2`, `clear/1`, `above/2` and `ontop/2`. A pruning operator removes rules that have not been used for 10 consecutive episodes.

The training set contains groups of 5 instances with respectively 3, 4, 6, 8 and 10 blocks, yielding a total of 25 training environments. The training set is created for every test separately to prevent learning a solution for a single configuration of the BLOCKS WORLD. A random BLOCKS WORLD generator is used to create the initial states, and it is ensured that all initial states are not goal states. The aspects for the main test set are summarized below. The test set consists of a combination of all values, and each test is executed 35 times (yielding a total number of $2 \times 6 \times 4 \times 3 \times 3 \times 35 = 15120$ tests).

Test aspect	Alternatives
policy application	probabilistic, deterministic
population size	$n_{chi}/n_{mut} \in \{20/40, 30/30, 40/20, 40/80, 60/60, 80/40\}$
episodes	20, 50, 100, 200
reproduction	$p_c \in \{0.6, 0.75, 0.9\}$ $p_m \in \{0.1, 0.2, 0.3\}$

Alternatives Test Set. The alternatives test includes a variety of alternatives in training set, fitness function, policy preservation and language bias. For these tests a combination of parameter values is used that have been deduced from the main test set. In addition to the `on(a, b)` task used in the main test set, GREY is tested on the `stack` and `unstack` tasks.

For both tasks the maximum number of steps needed by an optimal policy to solve the task is $n - 1$. Again, GREY is allowed $t_{max} = 2n$ steps to solve the problem – the training environments are created similarly as with the `on(a, b)` task in the main test set. Performance of alternatives is compared using the goal reaching performance (see Equation 5.3). Again, the average over 35 runs is a test result for a specific parameter setting.

An early fitness function definition used in GREY was the squared value of the reward obtained, divided by $|\pi| \cdot t_\pi$. This represents a more straightforward relation between the rewards, number of steps and size of the policy than the one defined in Equation 5.5 which was used in the main test set. To find out whether the design of the fitness function has a substantial influence on the performance of the algorithm the second fitness function is tested as an alternative. In order to ensure preservation of the best policy of a generation, the individual with highest fitness is copied to the new generation. It takes the place of a mutant to keep the same generation size.

Furthermore, it is interesting to test the influence of the predicate bias. We test the consequences of the availability of only the optimal set of relations, redundant relations and useless relations. A more optimal set is created by only allowing the predicates used by the optimal policy (i.e. `clear/1` and `ontop/2`) for the `on(a, b)`. A second test is adding redundant predicates in the form of the `talleststack/1` predicate, which is of no use in the `on(a, b)` task. Finally, a useless relation is added by the predicate `norel/2`, which has no real implementation; it always resolves and therefore simply binds two blocks to its arguments.

5.4.3.1 RESULTS ON THE MAIN TEST

The objective of the main test set is to study which parameter settings provide the best performance of GREY. Additionally, a comparison between probabilistic action selection, deterministic action selection and an optimal policy can be made. Here, we highlight some of the main findings, and for additional results and graphs, we refer to (Muller, 2005).

In order to find the best reproduction rate settings, the goal-reaching performance of the best policies after 100 learning episodes were compared for each population size. The learning curves for different population sizes have a similar shape and influence of the reproduction rates on the performance of the system is not much correlated to population size and action selection type. The best performance is obtained with a mutation probability of $p_m = 0.2$ or $p_m = 0.3$. Apparently, a high number of mutations is needed to get good performance. This can be explained by the fact that the search space in a relational domain is relatively large, so many mutations are needed to get sufficient exploration. The best crossover rate alternates over all of the values $p_c = 0.6$, $p_c = 0.75$ and $p_c = 0.9$. The chosen crossover rate seems relatively unimportant and this can be taken as an indication that an evolutionary algorithm is robust in this context. A higher number of learning episodes almost always results in better performance, regardless of the population size and proportion (there is one exception for probabilistic action selection between 100 and 200 episodes). From these tests no conclusion of convergence after some number of episodes can be made. This number will differ in case of different tasks, since learning a good policy will take more time for more difficult tasks. The influence of the population proportion on the performance increases with the number of learning episodes. If the system is allowed a larger amount of learning episodes, the 1 : 2 proportion in children respectively mutants is outperformed by the other proportions 2 : 1 and 1 : 1. This has possibly to do with the

fact that crossover keeps building blocks within rules intact where mutation more easily destroys them. In the current set of reproduction methods, only mutation can destroy the functioning of a single rule, while crossover cannot.

Best performances are reached by the population with 60 children and 60 mutants and the population with 80 children and 40 mutants for both probabilistic and deterministic action selection. The best performance overall is reached by deterministic action selection, 200 learning episodes, a population of 80 children and 40 mutants and reproduction rates $p_c = 0.6$ and $p_m = 0.2$. The average goal-reaching performance of the best policy in these settings is 1641.69. This is a very good performance in comparison to an optimal policy, since it is close to the median of the performance interval of an optimal policy. The performances of the better settings after 100 and 200 episodes are increasingly close to the worst-case performance of an optimal policy. The following table presents the performances of deterministic action selection and probabilistic action selection in terms of goal-reaching performance, number of goals reached and policy size. The numbers show that deterministic policies outperform probabilistic policies. The numbers shown are averages over 35 tests of learning 200 episodes (80 children, 40 mutants, $p_c = 0.6$, $p_m = 0.2$).

Version	Goal-reaching perf.		Goals reached		Size best
	Maximum	Average	Maximum	Average	policy
Deterministic	1641.69	1170.50	23.34	18.31	5.17
Probabilistic	1197.07	822.56	21.00	17.95	24.69

The first two columns present the goal-reaching performance of the best policy and the average goal-reaching performances of all policies after learning ended. The next two columns show the values for the number of goals reached by the policies in the 25 test environments. The number in the third column tells how many goals are reached by the best policy after learning (averaged over 35 tests) and the fourth column tells how many goals are reached on average by the whole population. Of the 35 tests performed, 71.43% of the best policies of the deterministic version reached all 25 goals and 54.29% of the 35 best policies of the probabilistic version reached all 25 goals. Finally, the last column presents the average size of the best policy after learning 200 episodes. It can be concluded that the deterministic policies implicitly encourages compression in comparison to the probabilistic version.

To get an idea of the learning process, both variants of action selection have tried to learn the `on(a, b)` task using a fixed training set of 25 environments and the following parameter settings: 100 episodes, population of 80 children and 40 mutants, $p_c = 0.6$ and $p_m = 0.2$. The learning behavior of both types show that the final generation can vary heavily in performance. The probabilistic version displays more variance in performance. This makes sense, since if an optimal policy has not been learned, the same best policy might take more timesteps to reach the goal in different episodes; because of probabilistic action selection, it might choose the wrong action for a situation, while at other times it might take the right action. The performance of deterministic action selection is more constant at times. Nevertheless performance fluctuates, for which two explanations are possible. First, a single peak may be caused by the EA, which has the possibility of eliminating a good policy. Second, the action selection is not completely deterministic. The rule, which is chosen deterministically, may propose more than one action. Picking the action for execution from this set of proposed actions is done randomly, causing the same effect as in probabilistic action selection.

Learned Policies. The best policy (goal-reaching performance 1685.4839, 25 goals reached) of deterministic action selection in the test is

$$\begin{aligned} \text{clear}(X), \text{ontop}(X, Y), \text{block}(Y), \text{block}(Z) &\rightarrow \text{move}(X, \text{floor}) \\ \text{block}(b), \text{clear}(a), \text{above}(c, X), \text{above}(Y, X) &\rightarrow \text{move}(a, b) \end{aligned}$$

The first rule takes any clear block X which is on top of some other block Y (not the floor) and moves it to the floor. Only when all blocks are on the floor, the second rule moves block a onto block b . This policy is near-optimal; it is a good solution for the $\text{on}(a, b)$ task in any blocks world environment, but it does not take into account that only the blocks ontop of a and b need to be removed.

The best policy (goal-reaching performance 1741.9355, 25 goals reached) of probabilistic action selection is

$$\begin{aligned} \text{clear}(X), \text{ontop}(Y, Z), \text{on}(W, X), \text{block}(Z) &\rightarrow \text{move}(Y, X) \\ \text{not}(\text{ontop}(X, Y)) &\rightarrow \text{move}(a, b) \\ &\rightarrow \text{move}(a, b) \end{aligned}$$

It works similarly to the deterministic policy above. Regarding the first rule, X can only be substituted for the floor, since X is both clear and some W is on it. Again, some block Y on top of another block Z is moved to the floor. The body of the second rule never applies – $\text{ontop}(X, Y)$ is always true, since for instance there is always a block on top of the floor –, so this rule is never executed (it was signed ‘not executed for 7 time steps’). The final rule tries to move a on b .

The fact that this probabilistic policy has better performance than the deterministic policy can be explained by the very fact that it is probabilistic: it will try to execute the rule $\text{move}(a, b)$ more often. The deterministic version will always move all the blocks to the floor, while the probabilistic version does not necessarily ‘wait’ for this. When executed and both a and b happen to be clear, the goal state is reached. On the other hand, if the action is illegal, the state is not changed, which ‘costs’ a time step. Apparently, in the $\text{on}(a, b)$ task with this particular training set, the probabilistic version works slightly better.

5.4.3.2 RESULTS ON THE ALTERNATIVES TEST

For the alternatives test set, the following parameter settings are used: 100 episodes, 80 children, 40 mutants, $p_c = 0.6$ and $p_m = 0.2$ for every mutation. A learning time of 100 episodes is chosen, to allow improvements in performance for the alternatives – if the best setting of 200 episodes was used, less (or even no) improvement in respect to the original version may be possible. The other parameters are chosen as a result of the findings of the main test set.

In the unstack task, the best policies of 33 of the 35 tests (94.3%) were able to solve all 25 learning environments – two tests solved 20 respectively 22 environments, making the test average 24.77. The average number of goals reached was 19.91. A policy solving all 25 environments is

$$\text{clear}(X), \text{block}(X), \text{ontop}(X, Y), \text{block}(Y), \text{on}(Z, W) \rightarrow \text{move}(X, \text{floor})$$

This is an optimal policy, since it moves any clear block X not on the floor (i.e. on top of some other block Y) to the floor.

Learning the stack task, 62.9% of the tests was able to find a policy that solved all 25 test environments. The test average of the maximum number of goals reached was 19.26, while the average number of goals reached was 15.53. An example of a good policy learned in the stack task is

$$\begin{aligned} & \text{block}(W), \text{block}(X), \text{clear}(W), \text{clear}(X), \text{ontop}(X, X) \\ & \text{clear}(X), \text{talleststack}(X), \text{block}(X), \text{ontop}(Y, X), \text{on}(Y, Z) \rightarrow \text{move}(W, X) \end{aligned}$$

The policy moves a clear block W on top of the clear block X of the tallest stack – which means X is the top block of the tallest stack. This policy is near-optimal, since a substitution W/X is possible, resulting in the illegal action $\text{move}(X, X)$.

In this policy a shortcoming of the way policies are generated becomes clear: there is no way redundant predicates can be removed from a rule. Although this has no effect on the performance of the policy, it increases computation time. For instance, the $\text{ontop}(Y, X)$ predicate always substitutes Y for X (since $\text{ontop}(Y, X)$ is clear) and together with $\text{on}(Y, Z)$ is always true. These facts do have to be calculated every time all possible actions are searched for.

Design Alternatives. The alternatives for background bias, fitness function, selection and mutation have been tested with the deterministic version of GREY with following parameters: 100 episodes, 80 children, 40 mutants, $p_c = 0.6$ and $p_m = 0.2$ for every mutation. Two batches of 35 tests were run for the original version and one batch of 35 tests for every alternative. The average results of these tests are presented in the following table. The percentage of the 35 tests that produced a policy that could reach a goal state within the given time in every training environment is listed in the next table.

Version	Goal-reaching perf.		Goals reached		Size best policy
	Maximum	Average	Maximum	Average	
Redundant bias	1081.51	689.47	17.57	13.78	4.40
Original 1	1140.01	758.05	18.49	14.14	4.17
Original 2	1210.44	816.33	19.69	14.85	4.71
Useless bias	1231.97	820.13	19.91	15.08	4.40
Fitness	1301.83	847.92	20.74	15.46	4.49
Preservation	1401.93	936.37	21.77	16.38	4.89
Perfect bias	1504.42	941.54	22.14	16.54	4.71

The runs for the original version still vary little in results, meaning 35 tests does not completely suffice to get equal results for identical settings. However, the results of the original version differs sufficiently from the alternatives to make some conclusions. Percentage of tests that resulted in a learned policy reaching the goal in all 25 training environments for every design alternative:

Version	Percentage	Version	Percentage
Redundant bias	28.6%	Useless bias	54.3%
Fitness	48.6%	Preservation	62.9%
Original 1	51.4%	Perfect bias	74.3%
Original 2	54.3%		

The use of useless predicates as in the ‘useless bias’ version, where the `norel/2` was added, does not have any effect on the performance – the numbers are actually better than for both original versions, although not significantly. This can be explained by the fact that it always resolves – if the relation appears in the body of the rule, it has as much effect as if it were never there. Using only the predicates needed for the optimal policy does have effect on the performance. The version with this perfect bias (only `clear/1` and `ontop/2` as background knowledge) performs best of all. Reducing the number of background knowledge predicates decreases the search space, allowing for quicker discovery of optimal or near-optimal policies. Accordingly, the version with redundant bias in the form of the `talleststack/1` relation performs worse than the original version – this relation appeared frequently in the final policies of this alternative. These results show that the choice of available predicates has much influence on the learning rate.

The complex fitness function does not improve performance in comparison to the alternative fitness function. When comparing the goal-reaching performance with the fitness, the complex fitness function shows the same ascent as the less complex function (the difference between the two learning curves can be explained by the square in the function). However, this does not mean the fitness function has no influence on the performance of the evolutionary algorithm in general: both fitness functions show quick changes, so a function with less gradient may work better. Additionally, an observation about the size of the best policy can be made: it is more or less the same for all alternatives. Apparently, the number of rules in a deterministic policy is independent from the settings tested in the design alternatives.

5.4.3.3 DISCUSSION

We also performed experiments in the TIC-TAC-TOE domain and the FROGGER domain. FROGGER is an old computer game in which a frog has to cross a road, and where the road is occupied by a number of lanes filled with moving traffic. The goal of the frog is to reach the other side without being run over by a car. The results in FROGGER are encouraging (the best frog reached the other side of the road 17 times), but the variance in policy performance is large and more detailed experiments are needed. For TIC-TAC-TOE the predicate bias used was too limited to learn reasonable policies (i.e. it consisted only of the basic predicates and `line`).

We must stress here that GREY is a proof-of-concept approach, and only represents a general outline of an evolutionary policy search algorithm for RMDPs. We tested GREY on the same problems that early value-based approaches such as Q-RRL and TG focused on, and obtained acceptable results. The main result is that we have obtained these using a very simple and straightforward adaptation of a GA algorithm, working on first-order logical representations. The strong point of GREY is that it can be easily extended in orthogonal directions, for example by providing more focused refinement operators, a better fitness function definition, and a better sampling process. A large improvement is to be expected when the recombination operator could be avoided as much as possible. As with neural networks, for example, exchanging large chunks between solutions may render both children relatively useless, even though their parents may have good performance. A two-level evolution process similar in spirit to SANE (Moriarty and Miikkulainen, 1996), where one level consists of atoms (or rules) and another consists of policies, would be very interesting.

Above all, all recent accomplishments in the field of evolutionary optimization could be applied in this setting. One of the reasons for its performance on the simple BLOCKS WORLD problems is that by moving to a policy-based search, the solution space becomes very small and a heuristic search is apparently sufficient to learn a good or optimal policy. In its current form, GREY cannot scale up to complex BLOCKS WORLD problems such as used by the LRW-API approach (see later in this chapter).

5.5. A Survey of Policy-Based Model-Free Relational RL

Whereas in the CARCASS approach value functions are the main focus of learning – and where policies are generated from these value functions – GREY’s representation and algorithms revolved around *policies*. GREY is an instance of a general pattern in relational RL, which can be characterized as a standard *policy search* (see Section 3.7) process. One starts with a policy structure $\tilde{\Pi}^0$, generates samples by interaction with the underlying RMDP, generates a new abstract policy $\tilde{\Pi}^1$, and so on. The general structure (see also Equation 4.11 in Chapter 4) is the following:

$$\tilde{\Pi}^0 \xrightarrow{\mathbf{S}} \{\langle s, a, q \rangle\} \xrightarrow{\mathbf{I}} \tilde{\Pi}^1 \xrightarrow{\mathbf{S}} \{\langle s, a, q \rangle\} \xrightarrow{\mathbf{I}} \tilde{\Pi}^2 \longrightarrow \dots \quad (5.6)$$

An important difference with value learning is that no explicit representations of \tilde{Q} are required. At each iteration of these *approximate policy iteration* algorithms, the current policy is used to gather useful learning experience – which can be samples of state-action pairs, or the amount of reward gathered by that policy – which is then used to generate a new policy structure.

In general, policy-based approaches use a form of *production rule system* (PRS) (i.e. a decision list) to represent the policy, and structural learning is either focused on complete structures or individual rules. Currently there are three types of approaches that follow the general structure in the above. The first is based on evolutionary search (e.g. as in GREY), the second induces complete policy structures from sampled state-action pairs, and the third uses gradient descent learning approaches on fixed policy structures.

5.5.1 Evolutionary Policy Search

The first type of policy-based approaches are *evolutionary* approaches such as GREY (Muller and van Otterlo, 2005; Muller, 2005). Distinct features of these approaches are that they usually maintain a *population* (i.e. a set) of policy structures, and that they assign a single-valued *fitness* to each policy (or policy rule) to guide the learning process. The FOX-CS system (Mellor, 2005a,b, 2007), discussed in Section 5.3.2, is a Michigan-style classifier system approach in which evolution is applied to the individual rules. In contrast, GREY is a Pittsburgh-style GA, in which complete structures are evolved. There are several propositional approaches in RL that are e.g. based on various forms of classifier systems (e.g. see Lanzi, 2002, and Chapter 3) and (neuro-)evolutionary algorithms (see Section 3.6.2.2). Even complete state spaces – if very small – can be encoded on a chromosome to evolve policies for MDPs. However, for most problems some type of abstraction is required. Research in first-order domains has so far mostly been restricted to deterministic planning, but could possibly be applied to general RMDPs.

Gearhart (2003) employs GP in the real-time strategy FREECRAFT domain used by Guestrin *et al.* (2003a) (see next chapter). Results show that it compares well to Guestrin

et al.'s approach, though it has difficulties with rarely occurring states. Castilho *et al.* (2004) focus on STRIPS planning problems, but unlike GREY for example, each chromosome encodes a full plan, meaning that the approach searches in *plan space*. A crucial aspect of this approach is the initialization process in which knowledge of the plan graph is inserted into the initial population. Both Kochenderfer (2003) and Levine and Humphreys (2003) do search in policy space, and both use a GP algorithm. Levine and Humphreys learn decision list policies from optimal plans generated by a planner, which are then used in a *policy restricted* planner. Kochenderfer allows for *hierarchical* structure in the policies, by simultaneously evolving sub-policies that can call each other. Finally, Baum (1999) describes the HAYEK machines that use evolutionary methods to learn policies for BLOCKS WORLDS (see also Baum, 1998, 2004, for an extensive discussion of relations between EAs, RL and economic theory). Individuals in the population are condition-action pairs, and a value for this action is learned by temporal difference learning. When an action has to be taken, each individual can bid on getting the right to perform its action, using its value as a bid. Within this *economy* of individuals, each individual should learn an accurate value of its worth, and evolution ensures that low-valued individuals are discarded and high-valued ones are propagated through the population.

As with GREY, for most of these approaches it is not yet known how generally applicable they are. However, they do show that with only modest modeling efforts, the EA can find good or optimal policies (see also Kress and Seese, 2007, for an application of a system very similar to GREY that was used for *executable product models* in the context of business process management.). But, despite some successes on (mainly) deterministic planning problems, general EA's have a number of additional disadvantages when employed in (relational) RL problems (Moriarty *et al.*, 1999). For example, online learning (e.g. in a physical environment) is difficult due to the nature of the population-based learning of EAs. Other disadvantages are difficulties with handling *rare states* and the type of feedback: TD algorithms generally sustain knowledge of both good and bad actions, and try to learn values for all states, whereas EA's only tend to keep information on general, good behaviors. Also problematic is the analysis of the *convergence* and *optimality* of algorithms, which is much less developed than methods based on TD. On the other hand, most of these aspects are equally less well developed for most algorithms that incorporate both structure and parameter learning in (relational) RL.

5.5.2 Policy Search as Classification

A second type of approach uses ILP algorithms to learn the structure of the policy from sampled state-action pairs. This essentially transforms the RL process into a sequence of supervised learning tasks in which an abstract policy is repeatedly induced from a (biased) set of state-action pairs sampled using the previous policy structure. This setup provides more information about the quality of individual actions, and it is more stable in the structure learning phase (because it operates on a stable dataset), though here the challenge is to obtain good samples, either by getting them from optimal traces (e.g. generated by a human expert, or a planning algorithm), or by smart trajectory sampling from the current policy. Both types of approaches are PIAGET-3, combining structure and parameter learning in a single algorithm.

Early on, Lecoecue (2001) introduced a system similar to the value-based Q-RRL approach, except that it uses *first-order decision lists* instead of trees. The system assumes a

small domain in which a ground Q -value function can be learned (or generated by other means) after which a ground policy is computed. This policy – in the form of state-action pairs – is then used as input for the decision list learner FOIDL (Mooney and Califf, 1995). The goal is to induce a compact first-order policy representation able to generalize over states and actions. The system is tested in a *batch* and a *semi-batch* manner. In the first, a ground policy already exists and an abstract policy is induced. The second case alternates between a learning phase and a policy induction phase. After each learning phase, a decision list policy is induced which is then used to bias the next learning phase. At all times, the system keeps ground representations of the current value function and policy, and in addition an abstract policy representation. The abstract policy used for guiding the learning process generalizes over unseen states and actions, and therefore speeds up learning. The system is tested in a dialogue system application which was used in earlier work on RL. The model-based approach by Yoon *et al.* (2002), which is discussed in the next chapter, uses similar ideas to induce policies from optimal traces generated by a planner. The algorithm can be viewed upon as an extension of the work by Martin and Geffner (2000, 2004) and Khardon (1999b) to stochastic domains. Khardon (1999b,a) studied the induction of deterministic policies for undiscounted, goal-based planning domains, and proved general PAC-bounds on the number of samples needed to obtain policies of a certain quality. A difference between Khardon’s approach and that of Martin and Geffner is that while the former uses a Horn language, the latter employ a concept language.

The LRW-API approach by Fern *et al.* (2006, 2007) unifies, and extends, the aforementioned approaches into one practical algorithm. LRW-API is based on a concept language (taxonomic syntax), similar to Martin and Geffner (2000)’s approach, and targeted at complex, probabilistic planning domains, as is Yoon *et al.* (2002)’s approach. LWR-API shares its main idea of iteratively inducing policy structures (i.e. *approximate policy iteration*, API) and using the current policy to bias the generation of samples to induce an improved policy with, for example Lecoeuche (2001). Two main improvements of LRW-API relative to earlier approaches lie in the sampling process of examples, and in the bootstrapping process. Concerning the first, LRW-API uses *policy rollout* (Boyan and Moore, 1995) to sample the current policy. That is, it estimates all action values for the current policy for a state s by drawing w trajectories of length h , where each trajectory is the result of starting in state s , doing a , and following the policy for $h - 1$ more steps. Note that this requires a simulator²¹ that can be sampled from any state, at any moment in time. The *sampling width* w and *horizon* h are parameters that trade-off variance and computation time. Additionally, instead of using the estimates $Q^\pi(s, a)$ directly, so-called *Q -advantages* are computed that relate the Q -values of state-action pairs in two subsequent policies. In this way, learning is focused towards instances where large improvement over the previous policy is possible. Fern *et al.* (2006) extend Khardon (1999b)’s theoretical results to general reward functions in this new setting. A second main improvement of LRW-API is the bootstrapping process, which amounts here to *learning from random worlds* (LRW) (Fern *et al.*, 2004a). The idea is to learn complex problems by first starting on simple problems and then iteratively solving more and more complex problem instances. Each problem instance is generated by a *random walk of length* n through the underlying RMDP. For example, an initial state distribution delivers a starting state s and from there n random

²¹Note that this is different from most other relational RL systems that base their estimates only on the trajectories that are generated by running full episodes.

actions are taken until arriving at some random state s' . The problem instance resulting from that is to get from s to s' . By increasing n , increasingly more complex problems can be tackled. An earlier approach without the random world sampling used the FF-heuristic²² from the planning community in the policy-rollout phase as the value of the state at the end of the horizon (Fern *et al.*, 2003).

The LRW-API approach has been thoroughly tested in various planning problems, both probabilistic and deterministic (see also Yoon *et al.*, 2004; Fern *et al.*, 2004b). The experiments in BLOCKS WORLD are far more complex than in most other relational RL systems, and involve goal configurations specified in terms of all blocks. It performed well in the *international planning competition* (probabilistic track) in 2004 and was the first ML-based system in any planning competition.

5.5.3 Policy Gradient Approaches

A third type of approach focuses on parameter learning based on fixed logical structures, similar in spirit to the feature-based algorithms discussed in the first half of this chapter. As in the propositional setting (see Section 3.7), not many approaches for RMDPs use policy gradient (PG) techniques. Three *policy gradient* approaches have been proposed that search for parameters of a fixed policy structure (i.e. PIAGET-1), where this structure can be either given before learning, or generated from the domain structure or sampled experience (e.g. PIAGET-0).

Itoh and Nakamura (2004) describe an approach for partially observable RMDPs in which policies are represented as a relational decision list. The hand-coded policies make use of a memory consisting of a limited number of memory bits. The algorithm is tested in a maze-like domain where planning is sometimes useful and the problem is to learn *when* it is useful. This is done by treating the policy as stochastic where the probabilities for the policy rules are used for exploration and learned via gradient descent techniques. The related approach by Wang *et al.* (2008b) operates on MLN representations of the policy.

More recently, Gretton (2007a,b) developed the *Relational Online Policy Gradient* approach (ROPG) that learns *temporally extended* policies for domains with *non-Markovian* rewards. To this end, the policy language is extended to incorporate ideas from temporal logic. Policy gradient is employed using standard techniques (see Section 3.7) as in Itoh and Nakamura (2004)'s approach, but the interesting aspect is how the policy structures are generated. Gretton uses two types of algorithms. One is based on the API approach described in the previous section, employed on small problems. In contrast to this sample-based technique, the second algorithm that is used, generates policy rules from the domain model, using Gretton and Thiébaux (2004a)'s regression-based technique (see the next chapter), adapted to the non-Markovian setting. It is shown that both have their relative merits on e.g. execution speed, convergence properties and generalization capabilities. Despite slow convergence in general, the RPOG approach was experimentally shown to learn general policies in an elevator scheduling problem.

²²*Heuristics* can be used as a *measure of progress* for domains in which feedback is sparse. Random exploration in a state space containing only one goal state will not generate any feedback until the learner accidentally hits the goal state. Yoon *et al.* (2005) report on an algorithm for learning measures of progress in deterministic and stochastic planning domains. A compact representation is used that can represent a prioritized sequence of components of the heuristic function. In BLOCKS WORLD, these components can be e.g. how many blocks are already *well-placed*.

Recently, in the *non-parametric policy gradient* (NPPG) approach Kersting and Driessens (2008) applied the idea of *gradient boosting*, which has been used before for structured prediction. Given that it just *lifts*²³ the level at which gradients are computed, it can automatically be applied for both propositional and relational representations. The advantage here is that an existing tree learner such as TILDE can directly be plugged in the boosting framework and used for relational RL. One disadvantage is that many trees have to be induced even for very simple problems. No connections are made yet with the other policy gradient techniques, nor with other gradient boosting techniques.

5.6. Discussion

In this chapter we have presented CARCASS and GREY, two instances of two distinct classes of model-free approaches to RMDPs. In addition, we have surveyed algorithms of both classes, and described some relations between systems, based on their characteristics and based on historical connections. At this moment, it is not yet possible to perform a detailed evaluation of all systems to estimate their relative performance in terms of computational complexity, sample complexity, generality, speed, or applicability to arbitrary problems. The field is still very young, and methods differ highly in the representational and algorithmic aspects, and the domains that have been used in experimental evaluations. LRW-API seems to be the current state-of-the-art in relational RL, in terms of the complexity of the domains it can successfully handle. Still, from this chapter, we can distinguish the following important dimensions of model-free relational RL:

Expressivity and Smoothness of Representations Most systems are based on relatively simple representation language, though some (e.g. RPOG) use more expressive languages. The general challenge for value-based approaches is to represent the value function in sufficient detail. Doing that by attaching numbers to abstract state definitions – i.e. logical formulas that represent sets of states of the underlying RMDP – (e.g. as in Q-RRL and TG), requires a highly finegrained and complex formula-based representation of the value function (see also Chapter 6). Even for moderate BLOCKS WORLDS this quickly becomes too complex to learn in many methods. Several feature-based (e.g. RTD), kernel-based and distance-based approaches (e.g. KBR and RIB) and gradient-based approaches (e.g. RPOG) provide more *smooth* (and global) approximations and will, presumably, scale to bigger problems more easily, at the expense of being less comprehensible. An interesting alternative is provided by the TRENDI and RUTREE approaches that build up a coarse approximation of the value function using logical abstraction, and allow for finetuning using instance-based representations at the bottom of the trees. Such *hybrid* approaches may just be the right way to go in building approximate representations for RMDPs.

Feedback and Sampling The amount of feedback, and its form, differs much between approaches. Whereas evolutionary approaches may only need a single fitness value for a complete policy (e.g. GREY), other algorithms need to keep many kinds of statistics to perform structural induction (e.g. TGR). This is strongly connected to the particular way of sampling to get informative learning examples. Whereas some value-based

²³In the gradient boosting framework, features can amount to complete regression trees, and the overall approach can alternatively be seen as a *propositionalization* technique as in Section 4.1.3.3.

approaches use only the state-action-reward tuples seen along experienced traces, others (such as evolutionary approaches) have to perform a considerable amount of simulation of the current policy to evaluate their performance. The policy rollout technique used by LRW-API provides informative samples, but it assumes a stronger simulation model than most other approaches. Bounds on the amount of samples needed to guarantee a certain performance have only been provided for policy-based methods based on classification. In addition to generating typical RL experience samples, some approaches have used stochastic sampling of *structures* (e.g. RUTREE and the approach by Walker *et al.* (2004)). Generating *initial structures*, for example an initial policy structure to bias an evolutionary algorithm, has not yet been employed much. Additional mechanisms, such as the random worlds approach used by LRW-API are useful, and general, ways to guide the learner towards increasingly more complex problems (but see also *guidance and models* in Chapter 7).

Adaptability The ability to *adapt* is highly desirable in RL problems because of the apparent non-stationary nature of the learning problem. Whereas initial approaches in either value-based (e.g. Q-RRL) and policy-based (e.g. by Lecoecue (2001) and Yoon *et al.* (2002)) employed iterative procedures that induce similar structures again and again, more recent approaches apply incremental (e.g. TG or TRENDI) and even fully adaptable representations (e.g. TGR and RUTREE). Structure-adaptable representations have obvious advantages over static, or incremental approaches, though there is still much work to be done in order to obtain fully automatic input-output (black-box-like) learning machines such as MLPs are for the propositional setting. Evolutionary approaches use an alternative way to sustain useful structures found earlier, in the form of *inheritance* that stimulates the propagation of useful building blocks over the generations. Separating structure learning from parameter learning seems more efficient (e.g. LRW-API and RPOG) than doing both simultaneously.

Representational Bias The most pressing question in this chapter has been on *what* to represent *explicitly*. In the end, what is required is an abstract policy representation; value functions are merely an intermediate step to obtain good policies. Still, if adequate domain knowledge is available, it pays off to generate a representation a priori and use any of the fixed-representation algorithms to induce a good policy. Partial or approximate *models* can be quite useful when used appropriately, and they can often be learned with modest computational efforts (such as in CARCASS and MARLIE). If a full action model is known beforehand, other techniques become available (see Chapter 6). Such full models can also be learned (see Chapter 7), but this requires more efforts. Representing (and learning) value functions in first-order domains has as an advantage that it provides much more information about the relative quality of actions, than a policy for the same domain. Yet, value-based learning in worlds that can vary in their domain size, i.e. learning value functions for *families* of RMDPs, has an unclear semantics because it is not clear whether observed variance in values are due to the current abstraction level that may be too coarse, or due to differences between values obtained in different instances of the RMDP family. All approaches, however, are dependent on some form of bias, be it a hypothesis language, a given abstraction level, search biases and refinement operators, domain models or any kind of a priori knowledge that can help in learning.

Further investigation into differences between approaches in terms of this bias could shed light on their relative performance.

In the absence of a domain model, there are many routes to finding a good or optimal policy and we have described these in this chapter. The field of relational RL is relatively young, and it is too soon to prefer either value-based or policy-based methods. Both have their advantages and much depends on the available domain knowledge. Ultimately, convergence to an optimal policy may be less important than efficiently finding a reasonable approximation. In the next chapter we deal extensively with the setting in which a full domain model is available, and where optimal solutions can be guaranteed though also there, approximate techniques are widely used to cope with large problems.

CHAPTER 6

Model-Based Algorithms for Relational MDPs

This chapter studies cases of relational RL where the complete dynamics and reward distribution in a particular environment are known. In these cases, DP algorithms can be used to compute optimal value functions and policies directly using the model, without sampling or inductive leaps. In the first part, the general framework of intensional dynamic programming (IDP) is introduced. This framework formalizes the use of general structured representations in DP and shows under which circumstances DP algorithms can be implemented that compute at the level of structured representations of states, value functions and policies, without explicit state space enumeration. Various propositional, structured DP algorithms can be shown to be implementations of the general idea of IDP. Furthermore, it is shown that they compute the same value functions and policies as the basic DP algorithms in Chapter 2. The second half of the chapter introduces REBEL, the first implemented value iteration algorithm that can solve RMDPs. We show that REBEL is yet again an implementation of value iteration using structured representations, only now for relationally represented domains. Some new challenges are action parameterizations and possibly infinite (or indefinite) domain (i.e. state space) sizes. An experimental validation shows the viability of the approach and some new possibilities concerning the use of logic programming techniques for structural operations and the use of inductive techniques for policy induction are described too. The chapter ends with a complete survey of related work in model-based relational RL, embedded in the IDP setting.

THE AVAILABILITY OF AN ABSTRACT MODEL of the dynamics of an RMDP removes the necessity of sampling and taking inductive leaps to obtain an optimal policy. The previous chapter has explored the case where a model is not available and the current will cover the cases where one is. The central inferential mechanism in the model-free, relational setting is *induction*. These methods do not have enough information about the underlying RMDP and have to use *samples* and *inductive* steps to generate logical structures or to estimate parameters' values. In contrast, *model-based* methods in this chapter can in general be classified as *deductive*. That is, given an abstract model of the underlying RMDP, optimal value functions and policies can be computed by *logical deduction* using the model as a theory. Instead of (possibly unsound) generalization using induction, the model-based methods can be seen as employing *justified generalization*, assuming correctness of the given model. Although deduction is enough to compute exact,

optimal value functions and policies, the computational burden to perform all necessary (structural) operations is very high. For this reason, some approaches have introduced *inductive* procedures that give up on exactly representing value functions, thereby making the trade-off between optimality and both computational speed and the possibility of solving larger problems. Most of this chapter is concerned with exact methods, though we will show that most approximate methods are enhancements of the exact case.

Model-Based Algorithms. Model-based algorithms can be used when a full transition model and reward distribution are known. As such, we are in the same situation as *planning* approaches, yet with the added aspects of a probabilistic environment and the fact that optimality is expressed in terms of total reward obtained by a policy, i.e. a *universal plan*. We cast the problem into the MDP framework, and upgrade existing algorithms that were discussed in Chapters 2 and 3 by using first-order logical techniques from Chapter 4. This is the route we will take in this chapter. Along the same lines as we have done in Chapters 4 and 5, we take **i)** classical algorithms developed for MDPs, **ii)** draw inspiration from the way propositional abstraction is used in compact representations and algorithms, and **iii)** use these to make another step towards first-order domains, now for the model-based case. In general, we are left with the following two tasks:

- **Exploiting Structure in Representations:** Chapter 4 has covered at length much of the general representational aspects needed for structured, first-order representations of MDPs, in terms of RMDPs (Section 4.1.3.2) and FORMs (Section 4.5.1.1). Structured, (first-order) representations can exploit many types of structure found in (R)MDPs. The specifics of the representation language determine how compact descriptions can be made, and which types of aspects can be described.
- **Exploiting Structured Representations in Algorithms:** Structured representations give one the opportunity to compactly represent the problem, but they also give a base representation for algorithms operating on the description. Each specific representation comes with a number of structural operations and reasoning patterns (e.g. inductive and deductive). DP algorithms can exploit these and the structure captured in the model description when efficiently computing optimal, structured value functions and policies.

Exploiting Inherent Structure in RMDPs. As we have seen throughout the previous two chapters, various forms of structure can be found in MDPs. RMDPs are no different in that respect, and in addition, they offer extra possibilities for generalization over objects. The same five sources of structure we have identified in Chapter 3 can be found in RMDPs, and exploited using various implementations of the FORMs in Chapter 4.

Most types of abstraction and generalization can be understood in terms of *sets of states*, often denoted *state aggregations*, *regions*, *abstract states* or *blocks*. For example, a tree-based value function representation (such as the one in Figure 3.10) partitions the state space in a number of distinct state sets, clustering states that have the same value under a particular policy. Another example is given by the symmetries in the TIC-TAC-TOE example (see Figure 3.1) where states are clustered into sets if they are rotated versions of each other. Even neural networks can be seen as partitioning an n -dimensional state space. Each type of representation provides *syntactic descriptions* that have their semantics in the

underlying state sets they model. In the relational setting, state sets consist of first-order interpretations, and descriptions are often formed using logical languages.

In the model-free setting, state aggregations are usually obtained by inductive methods based on samples. In the model-based setting, they can be derived from the model description itself, using so-called *justified generalization*. Semantically, value functions and policies correspond to state space partitions, and actions correspond to (probabilistic) mappings between sets of states. In the first half of this chapter we define *set-based* DP in which the inherent set-based structure is made explicit. Later, these state sets are to be replaced by descriptions, in any particular representational format. Variations using efficient data structures, as well as various approaches to approximation, can be defined on the basis of exact state aggregations.

Exploiting Structure in Dynamic Programming. In addition to *representing* a sequential decision making problem in a structured form, structure can also be exploited in solution algorithms such as DP. For example, some algorithms for propositional, factored MDPs exploit compact tree-based and ADD-based representations to compute value updates over entire regions in the state space (see Section 3.5). Many types of representations, including relational, can be used in structured algorithms, and the specifics of the representational language for states and actions creates opportunities to solve problems that would not be possible to solve in a flat, atomic representation.

Let us assume a light-bulb world consisting of light bulbs of different color. The only (deterministic) actions available in this world are `turnOff/1` and `turnOn/1`. The `turnOn` action can switch one light in a state from an `off` to an `on` position, and `turnOff` does the opposite. Actions represent transitions between sets of states that differ in the number of turned on lights, by exactly one. Let the value function V be such that all states that have *exactly two green bulbs turned on* have a value 10 and all the rest have value 0. Even if we do not know which bulbs exist, we know that all possible states have been partitioned by V into two disjoint sets. Given that we know the values in V , and given the action dynamics, we can find a set of states that can reach the 10-valued states in one step. This set consists of all states in which **i)** there is either exactly one green bulb turned on, and `turnOn` is used to turn on a green light, or **ii)** there are exactly two green bulbs turned on, and any action will be performed, as long as it is not turning on or off a green light, or **iii)** there were three green lights on and one is turned off. This line of reasoning forms the core of *structured Bellman backups*. Based on a known partitioning of the state space by a value function, we can use the *inverse* of the actions to find sets of states of which we can compute a new value, based on a one-step look-ahead and the values in V . In probabilistic environments, things become more complicated, as different actions in a state may lead to different state sets in V . Still, the point is that based on the idea of *sets of states*, Bellman backups can use this structure, and in addition the structure inherent in the actions, to do *set-based* backups. In this chapter we will explore this idea in DP algorithms in full detail.

Furthermore, things get more interesting if – instead of the sets themselves – one uses structured *descriptions* that have these sets as their semantics. The idea of set-based DP can be extended to *intensional* DP by doing just that. Propositional-based descriptions such as trees can be used to create powerful DP algorithms that work at the level of propositional descriptions. Yet, one step further along these lines gets to DP algorithms that work in RMDPs. Using logical descriptions of states, actions, value functions and policies, we extend the ideas of set-based and intensional DP to develop a version of value iteration

that can solve RMDPs.

The central idea in all these approaches is that, *any structure* brought along in the way the problem is represented (i.e. set-based, propositional descriptions, or first-order logical descriptions), can be used and maintained in Bellman backups and DP algorithms, such that optimal value functions and policies can be computed *without explicit state space enumeration*. Remember from Chapter 4 that a characteristic pattern in model-based approaches is the following series of purely deductive steps:

$$\begin{array}{ccccccccc}
 \tilde{V}^0 \equiv \mathcal{R} & \xrightarrow{\text{D}} & \tilde{V}^1 & \xrightarrow{\text{D}} & \tilde{V}^2 & \xrightarrow{\text{D}} & \dots & \xrightarrow{\text{D}} & \tilde{V}^k & \xrightarrow{\text{D}} & \tilde{V}^{k+1} & \xrightarrow{\text{D}} & \dots \\
 \downarrow \text{D} & & \downarrow \text{D} & & \downarrow \text{D} & & & & \downarrow \text{D} & & \downarrow \text{D} & & \\
 \tilde{\Pi}^0 & & \tilde{\Pi}^1 & & \tilde{\Pi}^2 & & & & \tilde{\Pi}^k & & \tilde{\Pi}^{k+1} & &
 \end{array}$$

In the first-order logical setting, *deduction* (D) is employed to compute Bellman backups, i.e. reasoning from a logical representation of the value function representation using the (inverse of the) action rules as a kind of axioms. This idea bears similarities with *theorem proving* approaches to planning (e.g. Fikes and Nilsson, 1971). The idea is that the model description (e.g. a FORM) contains all the information to *deduce* the optimal value function V^* from the initial value function $V^0 = R$. In Section 6.3 we will describe a concrete implementation of these ideas, using probabilistic, first-order STRIPS and *conjunctive logic*.

Structured Bellman backups in any representational format, compute both structures and parameters and are classified as PIAGET-3. In addition to *representational approximations*, the above structure lends itself for *algorithmic approximations* as well. For example, one can restrict the number of deduction steps, thereby trading structural completeness for computation speed.

Goals and Outline of this Chapter. The general topic of this chapter is model-based algorithms for first-order domains. More specifically, there are four main contributions that can be distinguished.

First of all, we describe how to find structure in general MDPs, both in representations and in the DP algorithms that compute optimal policies. In four steps (Sections 6.1.1 to 6.1.4) we develop a Bellman backup operator that operates entirely at the level of *sets* of states, and use it in a set-based DP algorithm. Set-based DP can be shown equivalent to classical, state-based DP. Second, in Section 6.1.5 we employ the set-based DP framework as a starting point for *intensional* DP. By replacing sets by *descriptions* that compactly *represent* these sets, it is possible to perform DP algorithms using general representational formats. For this purpose we develop so-called *state description languages* in Section 6.1.5.2. Again, we derive a new Bellman backup operator, yet now in terms of structured representations, in Section 6.1.5.3. IDP formalizes the common ground between many existing structured DP algorithms (see Section 6.1.5.4). Third, and most importantly, we introduce REBEL¹ in Section 6.3. The IDP framework turns out to be general enough to be implemented in relational domains too. REBEL implements a Bellman backup operator which is then used for value iteration for RMDPs. We introduce a relational state description language in Section 6.2, based on probabilistic STRIPS representations described in Chapter 4. Experimental validation (Section 6.3.5) and some extensions (Section 6.4)

¹The name REBEL is an abbreviation of *relational Bellman operator*.

show REBEL’s potential. The fourth goal of the chapter is to provide a survey (Section 6.5) of other existing methods that share the same common ground as REBEL, rooted in IDP. Section 6.5.1 focuses on such *exact* methods. Yet, other model-based techniques that incorporate representational or parameter approximations, inductive techniques or any other means, exist, and we will describe these in Section 6.5.2.

6.1. Intensional Dynamic Programming in Five Easy Steps

Dynamic programming (DP) is a general term denoting a rich variety of solution techniques that are based on the availability of a transition and reward model of the environment, bootstrapping, the Bellman equations and full value backups. In Chapter 2, and more specifically in Section 2.5, we have described *state-based* formulations of DP algorithms such as value iteration and policy iteration. In Chapter 3 we have outlined various forms of abstraction, and in Section 3.5 we have seen some algorithms that employ compact representations in structured algorithms.

Structured representations and DP algorithms have been described in the literature mostly in the context of specific KR formalisms. In the current section we show what all these methods have in common and how structured algorithms *using any particular representational formalism* can be understood in terms of classical, state-based DP. More specifically, we show that even DP algorithms for first-order domains can employ exactly the same methodology, once we have lifted the representational, reasoning and action formalisms to a first-order context. In Chapter 4 we have seen that many more advanced algorithms that might be of use for MDPs can be understood as lifted version of classical algorithms, and this view will be strengthened by the methods developed in this chapter, in particular when we introduce REBEL in Section 6.3.

Our approach consists of five steps in which we will transform classical DP into *intensional dynamic programming*. Each step introduces several new representational and algorithmic aspects that together form the constituents of STEP V, in which we provide an *intensional* framework for DP. Intensional in this context means that a KR framework is used to pose a sequential decision making problem and algorithms work directly on the KR level to compute value functions and policies. This, in turn, enables to work at the level of *abstract states* without the need to perform computations over individual states. As we have argued in Chapter 3, this is vital for large MDPs.

The assumptions of classical DP, with which we start in this section, are: **i)** the MDP is discrete and all states and actions are known, **ii)** the reward function R is represented as a table with an entry for each state $s \in S$, **iii)** the transition function is represented by $|A|$ matrices, each containing $|S|^2$ entries for transition probabilities, **iv)** state value functions V^t in each DP iteration t are stored in a table of size $|S|$ and state-action value functions Q^t are stored in a table of size $|S| \times |A|$, and **v)** policies π^t are stored in a $|S|$ size table. In the coming sections we will change the representational framework to sets, and later to intensional descriptions of these sets.

So far what concerns the representational aspects. As we have explained in Chapter 2, DP algorithms use *backup operators* that compute values for single states such that complete *sweeps* through the state space are needed. Together with the representational upgrades we provide, we will upgrade the algorithms, and more specifically the backup operators, to work with sets, and later with intensional descriptions of these sets. We focus

on *exact* methods, i.e. for which it can be shown that they compute, modulo approximation bounds, the exact same value functions and policies as classical algorithms. Throughout the section we will use a discounted, infinite-horizon model of rewards, usually setting $\gamma = 0.9$ and a preview of the steps is outlined below.

- **STEP I: Classical Value Iteration.** This is the classical value iteration that computes a table-based value function using successive approximations.
- **STEP II: Value Functions in Set Notation.** In this step we replace tables by sets, but all computation happens at the level of individual states and transitions. Another important difference with STEP I is that we devise a *backward view* on value backups, i.e. a value backup consists of applying an *inverse* transition function from states with a known value and computing values for all states that can lead to those states. STEP II is restricted to deterministic settings.
- **STEP III: Set-Based Value Functions.** In this step we move to compact *set based* value functions and policies, in which states are aggregated when they share the same value or action. Value backups deliver sets of states, such that values for multiple states can be computed simultaneously. In this step, we see the first appearance of operations aimed at keeping the representation as compact as possible. Like STEP II, this step is limited to deterministic MDPs.
- **STEP IV: Set-Based Dynamic Programming.** Here we move to the general setting of probabilistic transitions, in the context of the set-based value functions and policies introduced in STEP III. A decomposition of probabilistic actions into a probability distribution over *deterministic outcomes* enables employing many of the techniques defined in STEP III. New tools are introduced for combining all the effects of probabilistic actions in set-based value backups.
- **STEP V: Intensional Dynamic Programming.** The final step is to move beyond explicit set-based representations and to introduce compact *intensional* descriptions that *describe* sets of states and actions. Several examples in the literature are discussed and it is shown how choices for specific KR *schemes* influence the complexity and compactness of DP algorithms. STEPS I–IV can be seen as working on a *semantic level*, i.e. directly in terms of the MDP and sets of states and actions, whereas STEP V is all about *syntax*, i.e. KR systems that compactly represent MDPs and algorithms that make use of this KR level.

6.1.1 STEP I: Classical Dynamic Programming

We start our discussion with classical DP algorithms, with a special focus on *value iteration* (VI) (Bellman, 1957). In Chapter 2 we have discussed this algorithm among several other methods to compute the (optimal) value function V^* for a given MDP $M = \langle S, A, T, R \rangle$. Algorithm 12 outlines pseudo-code for computing the optimal value function for a given MDP up to a certain precision (determined by the parameter σ). Throughout this chapter,

we assume² that R is a state reward function. The algorithm computes the series:

$$R = V^0 \longrightarrow V^1 \longrightarrow V^2 \longrightarrow \dots \longrightarrow \underbrace{V^k \longrightarrow V^{k+1}}_{\mathbf{B}^*} \longrightarrow \dots \longrightarrow V^* \xrightarrow{\text{convergence}} \quad (6.1)$$

The common way to look at this series, is to see it as a number of successive approximations of the optimal value function V^* , called *successive approximation* (SA). However, for the purposes of this chapter, it will be more convenient to also see it as a series of approximations of k -steps-to-go value functions. $V^0 = R$ is the immediate reward function, i.e. the 0-steps-to-go value function. The 1-step-to-go value function V^1 is obtained by computing the value of doing one step and ending up with the values in V^0 . The 2-steps-to-go value function is obtained using the same procedure of applying one step, but now ending up with the values in V^1 , and so on. By this procedure, one can compute the series $V^0, V^1, V^2, \dots, V^*$, see Figure 6.1. Convergence of the algorithm means that eventually V^* is obtained, representing the value function for an optimal policy π^* . A policy π for

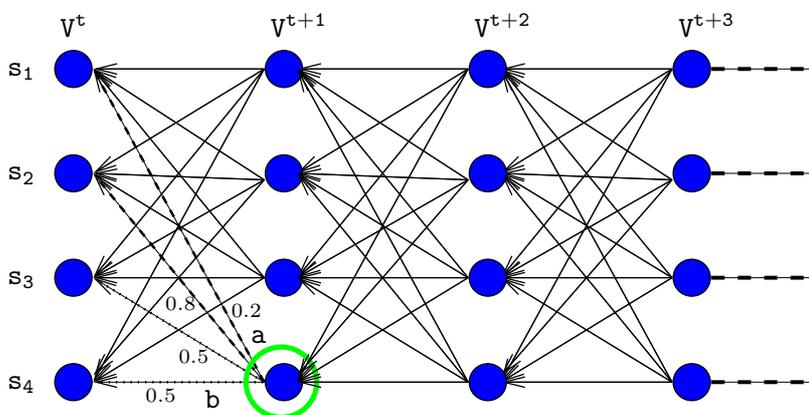


Figure 6.1: Computing V^{t+1} from V^t using the backup operator \mathbf{B}^* . Given a value function for a t -step planning horizon, the VI algorithm computes V^{t+1} thereby extending the current horizon with one additional step. The new value $V^{t+1}(s_4)$ for the highlighted state s_4 can be computed by as $R(s_4) + \gamma \max((0.2V^t(s_1) + 0.8V^t(s_2)), (0.5V^t(s_3) + 0.5V^t(s_4)))$.

all states $s \in S$ can, at each step in the approximation, be obtained from V^{k+1} , using $\pi(s) = \arg \max_{a \in A} \gamma \sum_{s' \in S} T(s, a, s') V^{k+1}(s')$. If VI is done by computing Q -functions as an intermediate step, a greedy policy is simply obtained as $\pi(s) = \arg \max_{a \in A} Q^{k+1}(s, a)$.

Each iteration in SA extends the k -steps-to-go horizon of the value function V^k to a $(k + 1)$ -steps-to-go value function V^{k+1} using the *backup operator* \mathbf{B}^* :

$$V^{k+1}(s) = (\mathbf{B}^* V^k)(s) = R(s) + \gamma \max_a \sum_{s' \in S} T(s, a, s') V^k(s') \quad (6.2)$$

This operator is obtained by turning the Bellman optimality equation (see Equation 2.3) into an update rule, and applying it to all states simultaneously³, using the most recently computed value function as the target. The steps inside the **repeat** loop of Algorithm 12

²Extensions to other types of reward functions are straightforward, but do require modifications to the algorithms and will be discussed when relevant.

³We assume that each V^n value function is a separate table, such that we do not use – possibly asynchronous – inplace updating, see also Section 2.5.2.1.

implement \mathbf{B}^* . Note that we compute Q -functions explicitly, as an intermediate step. If we look more closely at \mathbf{B}^* , we see that – although it is defined on whole value functions – the computation happens *state-wise*. In the following sections, our main purpose is to use compact representations to perform backups over complete *sets* of states.

Actually, the core of all DP algorithms is formed by turning Bellman optimality equations into backup operators that compute a more accurately approximated value function V^{k+1} from the current value function V^k . A backup operator uses the model (i.e. T and R), and possibly a policy π , to compute new estimates for state values. We can further distinguish two other backup operators that are of interest. The first is \mathbf{B}^a that backs up values according to a specific action a :

$$Q^{k+1}(s, a) = (\mathbf{B}^a V^k)(s) = R(s) + \gamma \sum_{s' \in S} T(s, a, s') V^k(s') \quad (6.3)$$

This backup may not be of much interest for use throughout the whole state space, but it can be used to backup values to regions in the state space where this action is chosen. The second backup operator \mathbf{B}^π backs up values according to a given policy π . This backup operator is used in e.g. policy iteration when computing V^π for a given policy π :

$$V^{k+1}(s) = (\mathbf{B}^\pi V^k)(s) = R(s) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V^k(s') \quad (6.4)$$

This operator is the same as the previous one \mathbf{B}^a , where the actual action a chosen is determined by the policy action $\pi(s)$ in each state $s \in S$. And in analogue to \mathbf{B}^* , the operator \mathbf{B}^π is a Bellman equation turned into an update rule. Note that \mathbf{B}^* maximizes

Algorithm 12 STEP I: **classical value iteration for a discrete MDP** $M = \langle S, A, T, R \rangle$. Transition and reward functions, value functions and policies are all represented as tables. The lines 5–8 implement \mathbf{B}^* , whereas line 9 checks for convergence.

```

1: [STEP I] : Classical Value Iteration
2:  $V^0 = R$ 
3: repeat
4:    $\Delta := 0$ 
5:   for each  $s \in S$  do
6:     for each  $a \in A(s)$  do
7:        $Q^{k+1}(s, a) := \sum_{s'} T(s, a, s') \gamma V^k(s')$ 
8:        $V^{k+1}(s) := R(s) + \max_a Q^{k+1}(s, a)$ 
9:        $\Delta := \max(\Delta, |V^{k+1} - V^k(s)|)$ 
10:     $k := k + 1$ 
11: until  $\Delta < \sigma$ 
    
```

over all actions, and can be seen as a composition of \max and \mathbf{B}^a , which will be made more explicit in the following sections. \mathbf{B}^* functions as a *contraction mapping* on the value function. If we let π^* denote the *optimal* policy and V^* its value function, we have the relationship (fixed point) $V^* = \mathbf{B}^* V^*$ where $(\mathbf{B}^* V)(s) = \max_a (\mathbf{B}^a V)(s)$. If we define $Q^*(s, a) = \mathbf{B}^a V^*$ then $\pi^*(s) = \pi_{\text{greedy}}(V^*)(s) = \arg \max_a Q^*(s, a)$. Algorithm 12 starts with an arbitrary value function V^0 after which it iterates $V^{k+1} = \mathbf{B}^* V^k$ until $\|V^{k+1} - V^k\|_S < \sigma$, i.e. until the distance between subsequent value function approximations is small enough.

One simple criterion is $\sigma = \frac{\epsilon(1-\gamma)}{2\gamma}$, such that V^{k+1} is within $\frac{\epsilon}{2}$ of V^* at any state, and that a greedy policy induced from V^{k+1} is ϵ -optimal (i.e. its value is within ϵ of V^* , see further Bertsekas and Tsitsiklis, 1996).

Summarizing, VI computes successive approximations of V^k , starting with V^0 , by the application of backup operators, and converges because these operators implement a contraction mapping.

A Note on Policy Iteration. As mentioned, we focus on VI as one particular DP algorithm. However, for each of the five steps we develop, we will mention how to adapt the technical machinery to other algorithms such as policy iteration. In this first step, a simple modification to Algorithm 12 is the following. First, we assume there is a current policy π^k (where the initial policy π^0 may be chosen at random). Instead of computing the values $V^{k+1}(s)$ for each state $s \in S$ using lines 7 and 8, we simply compute $V^{k+1}(s) = \mathbf{B}^{\pi^k} V^k$ (see Equation 6.4) for all states. In other words, the value function V^π of a fixed policy π satisfies the *fixed point* of \mathbf{B}^π as $V^\pi = \mathbf{B}^\pi V^\pi$. This way, an algorithm similar to Algorithm 12 can be used to compute V^{π^k} , and from this a new (greedy) policy π^{k+1} can be derived. Following Section 2.5, this two-step process is repeated until the policy is stable.

6.1.2 STEP II: Replacing Tables by Sets

As a first step towards intensional DP we modify both the representational and algorithmic aspects of the previous algorithm. Concerning the first, we replace the lookup tables storing value functions and policies by *sets*. This is not yet an advantage over the table-based approach, but it allows us to introduce new concepts and notation. To focus on this step, we restrict our approach in this (and the next) step to *deterministic* MDPs. Let us first define value functions and policies *in set notation*.

DEFINITION 6.1.1 ▶ Let $M = \langle S, A, T, R \rangle$ be an MDP. A **state value function** $V = \{\langle s, v \rangle\}$ **in set notation** is a partial relation of type $V \subseteq S \times \mathbb{R}$. For all states $s \in S$; $V(s) = v$ if $\langle s, v \rangle \in V$. A **state-action value function** $Q = \{\langle s, a, q \rangle\}$ **in set notation** is a partial relation $Q \subseteq S \times A \times \mathbb{R}$. For all states $s \in S$ and actions $a \in A$; $Q(s, a) = q$ if $\langle s, a, q \rangle \in Q$. A **policy function** $\pi = \{\langle s, a \rangle\}$ **in set notation** is a partial relation of type $\pi \subseteq S \times A$. For all states $s \in S$, $\pi(s) = a$ if $\langle s, a \rangle \in \pi$.

Note that we now use *set membership* (i.e. \in) to access individual entries. Also note that in value functions we assume that for each state (resp. state-action pair) there exists at most one element representing its value⁴. For policies, we allow multiple $\langle s, a \rangle$ pairs for different, but equally good, actions a . Representing value functions and policies as sets requires that we define new operations to perform summation and maximization.

DEFINITION 6.1.2 ▶ Let V_1, V_2 be state value functions and let Q be a state-value function.

⁴This is just for ease of explanation at this point. In Chapter 4 we have seen various forms of (relational) abstraction such as decision lists, in which the semantics was given by an operational definition that takes into account the order of partitions. Here we can use similar operational mechanisms to deal with incomplete specifications.

$$\begin{aligned}
 V_1 \ominus^1 V_2 &= \{s, v - v' \mid \langle s, v \rangle \in V_1 \wedge \langle s, v' \rangle \in V_2\} \\
 V_1 \oplus^1 V_2 &= \{s, v + v' \mid \langle s, v \rangle \in V_1 \wedge \langle s, v' \rangle \in V_2\} \\
 \max^1 Q &= \{\langle s, a, q \rangle \in Q \mid \neg \exists \langle s, a', q' \rangle \in Q : (q' > q)\} \\
 \max_A^1 Q &= \{\langle s, v \rangle \mid \langle s, a, v \rangle \in Q \wedge \neg \exists \langle s, a', q' \rangle \in Q : (q' > v)\}
 \end{aligned}$$

The difference between \max^1 and \max_A^1 is that the first maximizes a given Q -function Q , whereas the second returns the value function $V = \max_a Q$ by maximizing over actions. Note that \ominus^1 , \oplus^1 , \max^1 and \max_A^1 are defined in terms of operations on individual items.

In addition to the representational upgrade to value functions, we outline a new algorithmic element that will change the way value backups are computed. Here, we introduce a *backward view* on value computations that is needed to cope with *structured* representations for DP. The standard *forward view* of a Bellman backup operator is that we compute the new value $V^{k+1}(s)$ of a state $s \in S$ by a forward projection of the action results onto V^k , through the model. The only unknown value in a Bellman Equation such as Equation 2.3 is $V^{k+1}(s)$. The transition function is used in the normal way, planning one step ahead. Conceptually, this makes B^* actually an operator on V^{k+1} that is computed *using* V^k , instead of operating directly on V^k .

However, starting from this STEP II, we assume that each iteration of the algorithm starts with an arbitrary representation of the value function V^k , and that computation is entirely focused on computing a new value function V^{k+1} from V^k . To put it differently, at each iteration we do not⁵ have a representation of V^{k+1} that we can fill with values for each state. Instead, we have a, possibly structured, representation of V^k from which a new, also possibly structured, representation of V^{k+1} must be computed. This also means that one cannot take some state $s \in S$ and apply a forward projection through the model. The backward view is more clear in this respect, because it is defined directly on the current value function V^k . That is, instead of computing the value of a state by a forward propagation through the model, we pick a state s in the current value function V^k and use the *inverse* of the transition function to **i)** find those states s' that can reach s in one step by taking some action a , and **ii)** compute the value of such states s' using $V^k(s)$. Because we assume deterministic transitions here, we can simply compute $Q^{k+1}(s', a)$ as $R(s') + \gamma \cdot V^k(s)$. The value $V^{k+1}(s')$ is then obtained by maximizing over $Q^{k+1}(s', a)$ for all actions $a \in A$. The inverse transition function can be defined in terms of T .

DEFINITION 6.1.3 ► The **inverse** of the (deterministic) transition function T is $T^{-1}(s, a) =_{\text{def}} \{s' \mid T(s', a, s) = 1\}$. $T^{-1}(s, a)$ is called the **full pre-image** of s and a .

This explicit definition of the inverse transition function is a first example of a technique called *regression* (Waldinger, 1977, see also Section 3.5.2). The regression of a set C of conditions through an action a is the *weakest set of preconditions such that performing a will make C true*. In the limited context here, regression of a state $s \in S$ through an action $a \in A$ is equal to $T^{-1}(s, a)$. In the following paragraphs we will encounter increasingly more complex versions of regression in more detail. Summarizing, we can distinguish the following two views:

- **Forward View:** Computing the value $V^{k+1}(s)$ of state $s \in S$ amounts to finding

⁵The difference is even more evident, if we look at in-place updating in DP, which is common practice in practical implementations. Often there is only one table that represents *the* value function V .

those states that can be reached in one step, using their values to compute action values and maximizing those action values. That is, for each action $a \in A$, there is one state s' for which $T(s, a, s') = 1$. For each of these actions, one can compute $Q^{k+1}(s, a) = R(s) + \gamma \cdot V^k(s')$. Maximizing Q^{k+1} over all actions $a \in A$ yields $V^{k+1}(s)$.

- **Backward View:** The starting point is the value function V^k . For every state $s \in S$ in V^k and all actions $a \in A$ the inverse transition function T^{-1} is used to obtain those states $s' \in S$ such that applying a in s' yields s . For each such state-action pair $\langle s', a \rangle$ a value $Q^{k+1}(s', a) = R(s') + \gamma \cdot V^k(s)$ can be computed. Maximizing Q^{k+1} over actions finally yields V^{k+1} . The backward view can be seen as composed of a *regression* step (e.g. finding states to backup a value to) and a *value computation* step (e.g. computing the actual value).

This difference is mostly conceptual in the limited setting of STEP II. One technical difference however, is that in the forward view, the value $V^{k+1}(s)$ can be computed sequentially for all states, whereas in the backward view, first *all* action values for actions that reach the states in V^k must be computed. Only then, one can compute $V^{k+1}(s)$ for all states $s \in S$ by maximization. For set-based and structured representations of value functions in the next steps, the difference becomes more involved. The forward view is then no longer applicable, because the exact structure of V^{k+1} must be computed from the structure of V^k , such that no forward projection from V^{k+1} is possible. For this reason, the backward view will be used in the rest of this chapter.

PROPOSITION 6.1.1 ► The forward view and the backward view are equivalent.

Algorithm 13 STEP II: **value iteration in set notation.** It computes exactly the same value functions as Algorithm 12, which are now represented as sets. This algorithm is restricted to deterministic transition functions. Note that an additional difference is that the way value backups are computed, is via the inverse transition function T^{-1} . Lines 6–9 compute $B^a V^k$ for each action a , whereas lines 6–10 compute the full backup $B^* V^k$.

```

1: [STEP II] : Value Functions in Set Notation
2:  $V^0 = \{\langle s, v \rangle \mid s \in S, R(s) = v\}$ 
3:  $k = 0$ 
4: repeat
5:    $Q^{k+1} := \emptyset$ 
6:   for each  $\langle s, v \rangle \in V^k$  do
7:     for each  $a \in A$  do
8:       for each  $s' \in T^{-1}(s, a)$  do
9:          $Q^{k+1} := Q^{k+1} \cup \{\langle s', a, (\gamma \cdot v) \rangle\}$ 
10:   $V^{k+1} := R \oplus^1 \max_A^1 Q^{k+1}$ 
11:   $\Delta := \arg \max_{\langle, \Delta \rangle} |V^{k+1} \ominus^1 V^k|$ 
12:   $k := k + 1$ 
13: until  $\Delta < \sigma$ 
    
```

Algorithm 13 shows a VI algorithm that employs both the representational idea of set notation and the algorithmic change to a backward view on value backups. Here we have split the backup operators into two operations. Line 9 adds all states that can reach the

currently considered state s (i.e. the full pre-image of s and a , with a partial Q -value) to the Q -value function. Line 10 then maximizes the Q -value function and adds⁶ the state reward. Instead of a table, we now have an (unordered) variable-size set of state-value pairs. We separate the contribution of R from the Q -values for reasons that will be made clear in later steps.

Policy extraction⁷ is a simple operation in set notation. For any state $s \in S$ we have $\langle s, a \rangle \in \pi$ when $\langle s, a, q \rangle \in Q^{k+1}$ and there is no $\langle s, b, q' \rangle \in Q^{k+1}$ such that $q' > q$. This amounts to computing $Q^{\max} = \max^1 Q^{k+1}$ after⁸ which $\pi^{k+1} = \{\langle s, a \rangle \mid \langle s, a, q \rangle \in Q^{\max}\}$.

Goals vs. General Reward Functions. Algorithm 13 operates on MDPs with general reward functions. In Chapter 2 we have encountered *goal states* modeled as *absorbing* states. For any such state $s \in S$, $T(s, a, s) = 1$ for all actions a , and $T(s, a, s') = 0$ for all other $s' \in S, s' \neq s$. Furthermore, the value of goal states always⁹ is the value given by the reward function. Using goal-based reward functions has consequences for value backups, because of the special transition probabilities and rewards for goal states.

First, let $G \subseteq S$ be the set of goal states¹⁰. In the absence of a reward function, can the (incomplete) set of goal states G be transformed into a complete reward function using $R = \{\langle s, v \rangle \mid (s \in G \wedge v = v_g) \vee (s \notin G \wedge v = 0)\}$, where $v_g > 0$ is an arbitrary, positive reward. The easiest way to modify Algorithm 13 is to make a distinction between goal states and non-goal states in line 9. If $s' \notin G$ then nothing is changed, else line 9 is changed into $Q^{k+1} := Q^{k+1} \cup \{\langle s', a, R(s') \rangle\}$, i.e. the value of goal states is known and never updated. Much of this chapter is about general reward functions (so we omit¹¹ goal states in our algorithms), but in REBEL, we mainly use a goal-based setting.

State Space Refinement vs. Expansion. Instead of transforming an incomplete goal specification G into a complete reward function R , the type of backward algorithms we describe, can also be used directly on G . Incomplete reward functions are common in goal-based problems, where the aim is to reach one of the goal states, and (discounted) values of non-goal states merely reflect a relative *distance* to goal states.

In the context of this section we obtain an interesting result, which is that the state space is *gradually revealed* by successive iterations. Each iteration of the algorithm adds those states to the current value function that lie just one step away of the states we have already computed. Furthermore, in the deterministic case we consider here, the algorithm will converge¹² after only $N + 1$ iterations, where N is the maximum amount of steps

⁶Note that for state-action reward functions, a slightly modified add (\oplus^1) operator would be required for state-action value functions.

⁷It is always possible to derive a policy either from both the state value function (V) or the state-action value function (Q). The latter case is easier, because it does not require the use of a model in this step. During VI Q -functions are computed such that it is easier to derive the policy from Q .

⁸This is equal to the projection $\pi^{k+1} = Q^{\max}|_{s,a}$.

⁹In the case of state reward functions. When using state-action reward functions, the value backup needs to take this reward into account.

¹⁰A modification to more than one goal state set is straightforward.

¹¹In many cases, the algorithms can be modified quite simply by distinguishing between goal and non-goal states. An extra condition for value updates then becomes something like "if the pre-state is not absorbing (i.e. goal) then do the normal update, else keep current value (or take the value found in the initial reward function)". Depending on whether state value functions or state-action value functions are used, other details may vary too.

¹²The explanation is that we need N iterations to obtain all states that can be reached, and (possibly)

needed to reach a goal state from any non-goal state.

In order to ensure that the value of a state is always defined, one can add *default rules* to value functions and policies. For example, the rule $V(s) = v$ if $\langle s, v \rangle \in V^k$, and 0 otherwise. Later in this chapter we will see that such default rules can easily be used in the algorithm. Furthermore, state space expansion is much related to *reachability analysis* for MDPs (e.g. see Boutilier *et al.*, 1999).

Policy Iteration using Set Notation. In contrast to STEP I, a modification towards policy iteration requires a little more effort. Because of the backward view, we first look for those states that have to be updated, after which a value is computed. In order to modify the algorithm, we insert a state-action pair into the state-action value function *only when it is prescribed by the policy*. Let π^k be the current policy. The following line replaces line 9 in the algorithm¹³:

if $\pi^k(s') = a$ then $Q^{k+1} := Q^{k+1} \cup \{\langle s', a, (\gamma \cdot v) \rangle\}$

Algorithm 13 can now be used to compute V^{π^k} (i.e. to compute the evaluation step). The maximization step in line 10 is then no longer necessary. Once V^{π^k} is computed, the improvement step is to derive a greedy policy π^{k+1} . Policy iteration in this setting is computationally more demanding than in the previous algorithm, because all state-action pairs are considered, whereas only some state-action pairs will actually be used for the Q -function. This aspect will gain even more importance when we move towards more complicated algorithms in the next steps.

6.1.3 STEP III: Set-Based Value Functions

In the previous step we have introduced value functions in set notation, though storage of values was done at the level of individual states, having the same size as tables. In the current step we move to a representation where the aim is to store each distinct value only once. Value functions, and policies, are represented in terms of a number of sets, each sharing an equal value or an equal policy action. Value functions correspond to *partitions* of the state space whereas policies correspond to *coverings*. In addition, backups of values are computed in terms of sets of states too. This makes it possible to find complete blocks of states that all share the same value. A crucial difference with many other abstraction and aggregation methods (see Chapter 3) is that the aggregation we employ in this chapter does not affect the final solution in the ground state space. In many other cases, aggregation may break the Markov property and effectively transform the MDP into a POMDP (see Chapter 3). Here we employ so-called *justified generalization*, which means that states are only aggregated when we *know* that they share the same value or policy action. This knowledge can directly be derived from the model of the MDP.

one iteration more to find the best action in the states that were added the last, because these last states might be connected too. When the transition function is probabilistic (an aspect we will postpone until STEP IV) states that are already included in the current value function, will have their value updated in each iteration. This is influenced by discounting, probabilistic effects and (self-) loops (see also the examples further in this chapter). In those cases, the number of iterations needed will be larger than $N + 1$, and additional mechanisms are needed to expand the state space. This is because when the transition function is probabilistic, we need to alter Algorithms 15 and 16 to keep track of new states that are revealed by only one (probabilistic) effect of some action, otherwise they would be lost in the combination step.

¹³This can be called *regression through a policy*.

DEFINITION 6.1.4 ▶ Let \mathbb{Z} be a finite set. A **covering** of \mathbb{Z} is a set $\{\mathbb{Z}_1, \dots, \mathbb{Z}_n\}$ where $\mathbb{Z}_i \subseteq \mathbb{Z}$ (for all $i = 1, \dots, n$), $\bigcup_{i=1}^n \mathbb{Z}_i = \mathbb{Z}$. A **partition** of \mathbb{Z} is a covering $\{\mathbb{Z}_1, \dots, \mathbb{Z}_n\}$ with the additional constraint that $\mathbb{Z}_i \cap \mathbb{Z}_j = \emptyset$ ($i, j \in 1, \dots, n, i \neq j$). Each partition is a covering, but the converse does not necessarily hold.

The space of all partitionings (or coverings) of a state space can be structured along the lines of being more coarse or fine-grained, enabling to make comparisons.

DEFINITION 6.1.5 ▶ Let \mathbb{Z} be a finite set, and let \mathbb{X} and \mathbb{Y} be two partitionings of \mathbb{Z} . \mathbb{X} is a **refinement** of \mathbb{Y} , denoted $\mathbb{X} \ll \mathbb{Y}$, iff every $\mathbb{X}_i \in \mathbb{X}$ is a subset of some $\mathbb{Y}_j \in \mathbb{Y}$. If, in addition, some $\mathbb{X}_i \in \mathbb{X}$ is a proper subset of some $\mathbb{Y}_j \in \mathbb{Y}$ than \mathbb{X} is said to be **finer** (\ll) than \mathbb{Y} . The inverse of refinement is **coarsening** (\gg) and that of finer is **coarser** (\gg).

Now we can define *set-based* value functions and policies as follows:

DEFINITION 6.1.6 ▶ Let $M = \langle S, A, T, R \rangle$ be an MDP. A **set-based state value function** is a set $V = \{\langle \mathbb{S}_1, v_1 \rangle, \dots, \langle \mathbb{S}_n, v_n \rangle\}$ where $\mathbb{S}_1, \dots, \mathbb{S}_n$ form a partition of S and v_1, \dots, v_n are values. Furthermore, for any $s \in S$ it holds that $V(s) = v$ if $\langle \mathbb{S}_i, v \rangle \in V \wedge s \in \mathbb{S}_i$. A **set-based state-action value function** is a set $Q = \{\langle \mathbb{S}_1, a_1, q_1 \rangle, \dots, \langle \mathbb{S}_n, a_n, q_n \rangle\}$ where $\mathbb{S}_1, \dots, \mathbb{S}_n$ form a covering over S , each $a_i \in A$ ($i = 1 \dots n$), and q_1, \dots, q_n are values. Furthermore, for any $s \in S$ it holds that $Q(s, a) = q$ if $\langle \mathbb{S}_i, a, q \rangle \in Q \wedge s \in \mathbb{S}_i$. A **set-based policy** is a set $\pi = \{\langle \mathbb{S}_1, a_1 \rangle, \dots, \langle \mathbb{S}_n, a_n \rangle\}$ where $\mathbb{S}_1, \dots, \mathbb{S}_n$ form a covering over S and each $a_i \in A$ ($i = 1 \dots n$). Furthermore, for any $s \in S$ it holds that $\pi(s) = a$ if $\langle \mathbb{S}_i, a \rangle \in \pi \wedge s \in \mathbb{S}_i$.

Note that state-action value functions form coverings of the state space, but partitions of the state-action space. Furthermore, note that a policy can simultaneously represent multiple policies with the same value function. This, in turn, may render a policy non-deterministic and an additional selection mechanism (e.g. random) is necessary to choose a concrete action in a state s when there exist more than one action a for which $\pi(s) = a$.

In the remainder of this section we will omit *set-based* when we mention value functions or policies. As mentioned, a difference with the previous step is that the representation is more compact. However, we still can access individual states in value functions and policies. Operations on value functions make use of the fact that they consist of sets:

DEFINITION 6.1.7 ▶ Let V_1 and V_2 be value functions over \mathbb{S} . The following operations define **addition** and **subtraction** of value functions, resulting in value functions over S :

$$\begin{aligned} V_1 \oplus V_2 &= \{\langle \mathbb{S}_i \cap \mathbb{S}_j, r_i + r_j \rangle \mid \langle \mathbb{S}_i, r_i \rangle \in V_1 \wedge \langle \mathbb{S}_j, r_j \rangle \in V_2\} \\ V_1 \ominus V_2 &= \{\langle \mathbb{S}_i \cap \mathbb{S}_j, r_i - r_j \rangle \mid \langle \mathbb{S}_i, r_i \rangle \in V_1 \wedge \langle \mathbb{S}_j, r_j \rangle \in V_2\} \end{aligned}$$

We can see that both operations result in finer partitions, due to the intersection operation. The operations are defined for both partitions and coverings.

DEFINITION 6.1.8 ▶ A **merge** operation on a value partition V gathers all parts that have the same value and joins them, resulting in a new value partition V' such that $V' \gg V$, and is defined as

$$\text{merge}(V) = \{\langle \bigcup_{\langle \mathbb{S}_i, r \rangle \in V} \mathbb{S}_i, r \rangle \mid r \in \text{VALUES}\} \text{ where } \text{VALUES} = \{r \mid \langle \mathbb{S}_j, r \rangle \in V\}$$

A similar merge operation can be defined for Q -functions, merging blocks that have the same value *and* action. The merge operation is the first step towards KR operations neces-

sary to keep the representation compact. Without it, the value function would be shattered more after each iteration.

EXAMPLE 6.1.1 ▶ Let $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$. Let $V_1 = \{\langle\{s_1, s_2, s_3\}, 1\rangle, \langle\{s_4, s_5, s_6\}, 2\rangle\}$ and $V_2 = \{\langle\{s_1, s_6\}, 8\rangle, \langle\{s_2, s_3, s_4, s_5\}, 9\rangle\}$. Adding both value partitions results in $V_1 \oplus V_2 = \{\langle\{s_1\}, 9\rangle, \langle\{s_6\}, 10\rangle, \langle\{s_4, s_5\}, 11\rangle, \langle\{s_2, s_3\}, 10\rangle\}$. The result contains two blocks having value 10. The reduced partition is $\text{merge}(V_1 \oplus V_2) = \{\langle\{s_1\}, 9\rangle, \langle\{s_2, s_3, s_6\}, 10\rangle, \langle\{s_4, s_5\}, 11\rangle\}$. Note that $\text{merge}(V_1 \oplus V_2) \succeq V_1 \oplus V_2$.

DEFINITION 6.1.9 ▶ A **maximization** operator takes a state-action value partition and identifies the highest state value throughout the whole space, and is defined as:

$$\begin{aligned} \max V &= \{\langle\mathbb{S} - H, v\rangle \mid \langle\mathbb{S}, v\rangle \in V, \mathbb{S} - H \neq \emptyset \\ &H = \bigcup\{\mathbb{S}' \mid \langle\mathbb{S}', v'\rangle \in V, \mathbb{S} \cap \mathbb{S}' \neq \emptyset, v' > v\}\} \end{aligned}$$

Maximization on Q -coverings is done by:

$$\begin{aligned} \max Q &= \{\langle\mathbb{S} - H, a, q\rangle \mid \langle\mathbb{S}, a, q\rangle \in Q, \mathbb{S} - H \neq \emptyset \\ &H = \bigcup\{\mathbb{S}' \mid \langle\mathbb{S}', a', q'\rangle \in Q, \mathbb{S} \cap \mathbb{S}' \neq \emptyset, q' > q\}\} \end{aligned}$$

The operation \max_A over Q -partitions is the same as \max , except that it discards the action (as in Definition 6.1.2), effectively turning the end result into a state value partition.

Maximization is slightly more complex than in the previous step¹⁴. However, there are much fewer blocks in the set-based approach such that fewer, yet more complex, computations are needed. The trade-off between compact representations (i.e. fewer components) and computational aspects is a core topic later in this section.

EXAMPLE 6.1.2 ▶ Let $S = \{s_1, s_2, s_3\}$ and $A = \{a, b, c\}$. Let $Q = \{\langle\{s_1, s_2, s_3\}, a, 10\rangle, \langle\{s_1\}, b, 5\rangle, \langle\{s_2, s_3\}, b, 15\rangle, \langle\{s_1\}, c, 15\rangle, \langle\{s_2\}, c, 15\rangle, \langle\{s_3\}, c, 7\rangle\}$. The maximization of Q results in $\max Q = \{\langle\{s_2, s_3\}, b, 15\rangle, \langle\{s_1\}, c, 15\rangle, \langle\{s_2\}, c, 15\rangle\}$. A (merged) state value function obtained by maximizing Q is simply $V = \{\langle\{s_1, s_2, s_3\}, 15\rangle\}$. Note that we omit the reward contribution here. A policy derived from Q is $\pi = \{\langle\{s_1, s_2\}, c\rangle, \langle\{s_2, s_3\}, b\rangle\}$.

Maximizing a Q -value function essentially corresponds to deleting state-action pairs that are *dominated* in terms of value by other state-action pairs. For example, if state s is covered by both $\langle\mathbb{S}, a, q_1\rangle$ and $\langle\mathbb{S}', b, q_2\rangle$ and $q_1 > q_2$ then s will be removed from \mathbb{S}' in the maximization operation.

Now that we have covered all representational issues, we need one additional tool. To obtain the *set* of states that can reach states in a block of the current value partition, we extend the inverse transition function accordingly.

DEFINITION 6.1.10 ▶ The **set inverse** T_S^{-1} of the (deterministic) transition function T is defined as $T_S^{-1}(\mathbb{S}, a) = \{\langle s', a \rangle \mid s \in \mathbb{S} \wedge T^{-1}(s) = \langle s', a \rangle\}$. When $T_S^{-1}(\mathbb{S}, a) = \mathbb{S}'$, then \mathbb{S}' is called the **full block pre-image** of \mathbb{S} and a .

The set inverse of T is used in line 8 of Algorithm 14 (see also Figure 6.2) to compute a complete block of states to be inserted in Q^{k+1} . Note that, although T_S^{-1} is defined over

¹⁴A practical, general procedure for maximization consist of first sorting the blocks on their value, followed by a double loop through the blocks to compute intersections. This makes the algorithm quadratic in the number of blocks.

sets of states, it is still defined in terms of individual state-action-state transitions (i.e. using T^{-1}). This will change in STEP V. The set inverse of the transition function makes

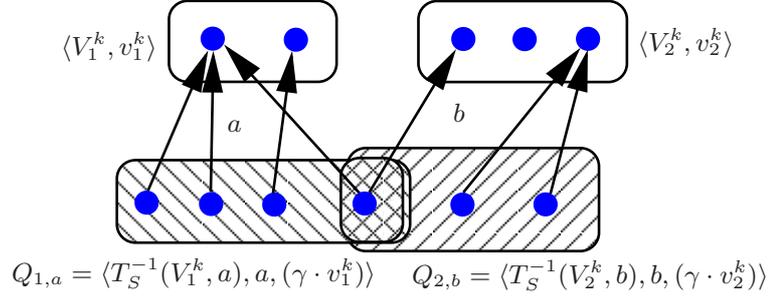


Figure 6.2: Set-based regression of a value function. Parts $\langle V_1^k, v_1^k \rangle$ and $\langle V_2^k, v_2^k \rangle$ are both in the set-based value function V^k . Shown is the result of regressing V_1^k through action a and V_2^k through action b . Both $Q_{1,a}$ and $Q_{2,b}$ are to be included in Q^{k+1} in line 8 of Algorithm 14. Note that the intersection of $Q_{1,a}$ and $Q_{2,b}$ is not empty. For the state s in that intersection it holds that taking action a in s will result in a transition to a state in V_1^k , whereas b will transition to a state in V_2^k .

Algorithm 14 STEP III: value iteration using set-based value functions for deterministic MDPs. Lines 6–8 compute $B^a V^k$ for each action a , whereas lines 6–9 compute the full backup $B^* V^k$. Line 10 merely optimizes the compactness of V^{k+1} by coarsening it.

-
- 1: [STEP III] : **Deterministic Transitions and Set-Based Value Functions**
 - 2: $V^0 = R = \{\langle V_1^0, v_1^0 \rangle, \dots, \langle V_n^0, v_n^0 \rangle\}$
 - 3: $k = 1$
 - 4: **repeat**
 - 5: $Q^{k+1} := \emptyset$
 - 6: **for each** $\langle V_i^k, v_i^k \rangle \in V^k$ **do**
 - 7: **for each** $a \in A$ **do**
 - 8: $Q^{k+1} := Q^{k+1} \cup \{T_S^{-1}(V_i^k, a), a, (\gamma \cdot v_i^k)\}$
 - 9: $V^{k+1} := R \oplus \max_A Q^{k+1}$
 - 10: $V^{k+1} := \text{merge}(V^{k+1})$
 - 11: $\Delta := \arg \max_{(\cdot, \Delta)} |V^{k+1} \ominus V^k|$
 - 12: $k := k + 1$
 - 13: **until** $\Delta < \sigma$
-

it possible to distinguish between two types of operations in Bellman backups, being a *structural* part and a *parameter* part.

- The **structural part** consists of computing the state space partitioning present in value functions. Based on the partition in V^k , the set inverse T_S^{-1} determines which distinctions must be made, and thus, which states are to be put together.
- The **parameter part** is concerned with attaching values to the partitioning. Each abstract state in V^{k+1} gets its value based on V^k , R and γ , as usual.

In this chapter we mostly concentrate on algorithms that combine structural and parameter computations. Nevertheless, it is also possible to first focus on the structural part and then learn parameters, for example by first generating an abstract state space and learning

values by sampling over this space (e.g. using model-free RL). The structural part in this kind of approaches is known as model-minimization (e.g. Givan *et al.*, 2003, and see further in Section 3.4).

It is easy to see that Algorithm 14 computes exactly the same value functions as Algorithm 13. The main difference lies in the more compact representation of the value functions. Summation and maximization operators now use intersections of sets, instead of individual lookups and comparison of states. As a consequence, Algorithm 14 converges to the optimal value function in the same way as Algorithm 13 and classical VI.

Similar to the previous algorithm, convergence can be split into a *structural* and a *parameter* part. The usual view, in line with classical VI, is to see it as computing a sequence of value functions $V^0 = R, V^1, \dots, V^*$. But, in the context of set-based value functions, we can also see the sequence of value functions V^1, \dots, V^* as *refinements* of $V^0 = R$. Each iteration computes V^{k+1} as a refinement of V^k . Structural convergence then means that the state partitionings in the value functions V^k and V^{k+1} are the same. Parameter convergence is defined in the classical meaning on the state values themselves, i.e. once the structure of the partition remains the same during iterations, Algorithm 14 degenerates to classical VI over sets of states. The algorithm can also be modified to work with incomplete reward function specifications in the same way as was described in the previous section, i.e. by defining default rules for set-based value functions and policies. In that case, new abstract states are generated along the way and the state space is gradually revealed. In the second half of this chapter this proves useful in REBEL when the state space is not known beforehand, or even infinitely large.

A Note on Policy Iteration. Policy iteration in the setting provided in this third step can be defined analogously to the previous step. That is, T_S^{-1} is used to find states that lead into the current value partition and values are computed for them. The difference is now that these states are contained in blocks. The policy too is represented in terms of blocks. Each time a new block $\langle S, a, q \rangle$ is to be inserted into Q^{k+1} , it must be *intersected* with the policy, in order to keep only those state-action pairs for which the action a coincides with the action prescribed by the policy for those states. This makes policy iteration in this setting even more computationally demanding, because the intersection with the full policy is an expensive operation. Furthermore, it potentially shatters the Q -function further (i.e. it refines it), and increases the costs of merging afterwards.

6.1.4 STEP IV: Set-Based Dynamic Programming

In the previous two steps, we have assumed that all transitions were deterministic. We will see that extending the framework so far with probabilistic transitions can be done most easily on top of the algorithms for deterministic environments. The key insight is to decompose probabilistic transitions into a number of deterministic transitions. In Chapter 4 we have already seen that many probabilistic action languages use such decompositions.

In this section we first assume a general MDP definition with a standard state-based transition function T . To obtain for each action a set of deterministic alternatives, we construct a new, deterministic transition function \hat{T} . To do this, a special type of structure must be found in the transition function. For each syntactically distinct action, we assume that for all states the *probabilistic transition structure* (PTS) is the same.

DEFINITION 6.1.11 ▶ Let $M = \langle S, A, T, R \rangle$ be an MDP. The **probabilistic transition**

structure pts for any state $s \in S$ and action $a \in A$ is defined as the (sorted) list $\text{pts}(s, a) = [T(s, a, s') > 0, s' \in \mathbb{S}]$. The action set A is **pts-uniform** if for all $a \in A$ it holds that for all states $s, s' \in S$, $\text{pts}(s, a) = \text{pts}(s', a)$.

For any general MDP definition, the assumption that the action set is PTS-uniform can be enforced by introducing new action names until all actions are PTS-uniform, although this can increase the total number of actions considerably. This is not due to a particular type of algorithm, but it is clearly related to the amount of *structure* in the MDP. Most applications of structured representations for MDPs assume that actions have a relatively small number of outcomes. If the action set is PTS-uniform, the notion of a probabilistic action can be defined as follows:

DEFINITION 6.1.12 ▶ A **probabilistic action** a that is decomposed into a set of deterministic actions is denoted $\triangleright_a = [\langle a_1, p_1 \rangle, \dots, \langle a_m, p_m \rangle]$, where each a_i ($i = 1, \dots, m$) is a deterministic action, $\text{pts}(s, a) = [p_1, \dots, p_m]$, $0 > p_i \geq 1$ ($i = 1, \dots, m$), and $\sum_{i=1}^m p_i = 1$. The **decomposed transition function** \hat{T} is defined as follows for all states $s, s' \in \mathbb{S}$: $\hat{T}(s, a_i, s') = 1 \Leftrightarrow T(s, a, s') = p_i$, and $T(s, a_i, s') = 0$ for all other transitions.

The purpose of the newly defined transition function \hat{T} is to 'push' the action probabilities to a higher level, with as main effect that we are able to use the previous algorithms that were defined for deterministic settings. Structural operations (e.g. regression over deterministic transitions) and parameter computations (e.g. values and probabilities) can in this way, again, be separated. Let us first consider an example of how a probabilistic action can be decomposed.

EXAMPLE 6.1.3 ▶ Let $\mathbb{S} = \{s_1, s_2, s_3\}$ and let a be an action. Let a part of the transition function that specifies transition probabilities for action a be as follows: $T(s_1, a, s_2) = 0.6$, $T(s_1, a, s_1) = 0.4$, $T(s_2, a, s_3) = 0.6$, $T(s_2, a, s_1) = 0.4$, $T(s_3, a, s_2) = 0.6$, and $T(s_3, a, s_1) = 0.4$. We can see that $\text{pts}(s, a) = [0.4, 0.6]$ (for all s) and a is decomposed into $\triangleright_a = [\langle a_1, 0.6 \rangle, \langle a_2, 0.4 \rangle]$. The decomposed transition function \hat{T} for a is defined as $\hat{T}(s_1, a_1, s_2) = 1$, $\hat{T}(s_1, a_2, s_1) = 1$, $\hat{T}(s_2, a_1, s_3) = 1$, $\hat{T}(s_2, a_2, s_1) = 1$, $\hat{T}(s_3, a_1, s_2) = 1$, and $\hat{T}(s_3, a_2, s_1) = 1$. Furthermore, $\hat{T}(s_i, a_j, s_k) = 0$ for all other transitions (s_i, a_j, s_k) ($i, j = 1, 2, 3, k = 1, 2$).

In the previous two algorithms we have used an inverse transition function to obtain states for which we can compute a new value. For each deterministic outcome of a probabilistic action we use the set inverse of \hat{T} , denoted \hat{T}_S^{-1} , which is defined analogous to T_S^{-1} (see Definition 6.1.10). Whereas T_S^{-1} computes a full pre-image of a deterministic action effect, \hat{T}_S^{-1} computes only a *partial* pre-image with respect to a probabilistic action. That is, all states computed by $\hat{T}_S^{-1}(\mathbb{S}', a_i)$, are those states for which it is known that one particular effect a_i of action a (with known probability p_i) will result in a transition to one of the states in \mathbb{S}' .

As can be seen in Equation 6.2 the value backup of a state in the forward view consists of a summation of the values of 'next' states, weighted by the probability distribution over action outcomes. In the backward view, we first compute partial pre-images that contain states for which a partial Q -value has been computed based on one outcome of a probabilistic action. This value can be computed based on the pushed-out outcome probability, and the partial pre-image that is labeled with this particular outcome. This is

translated into the summation of the partial value of a number of pre-images $\langle S_i, a_i, q_i \rangle$ that together represent transitions for all of the component actions of one probabilistic action a . The intersection of these pre-images contains just those states for which the Q -value of action a is completely characterized as $\sum_i q_i$ (i.e. the summation in Equation 6.2). For an example see also Figure 6.3.

DEFINITION 6.1.13 ► The partial block pre-images $\langle S_1, a_1, q_1 \rangle, \dots, \langle S_n, a_n, q_n \rangle$, generated from all outcomes a_1, \dots, a_n of probabilistic action a , together form a full block pre-image of action a as $\langle \bigcap_{i=1}^n S_i, a, \sum_{i=1}^n q_i \rangle$. The **combination** of a set of partial block pre-images Q is defined as: $\text{combine}(Q) = \{ \langle \bigcap_{i=1}^n S_i, a, \sum_{i=1}^n q_i \rangle \mid \langle S_1, a_1, q_1 \rangle, \dots, \langle S_n, a_n, q_n \rangle \in Q, a \in \mathbb{A}, \triangleright_a = [\langle a_1, p_1 \rangle, \dots, \langle a_n, p_n \rangle] \}$

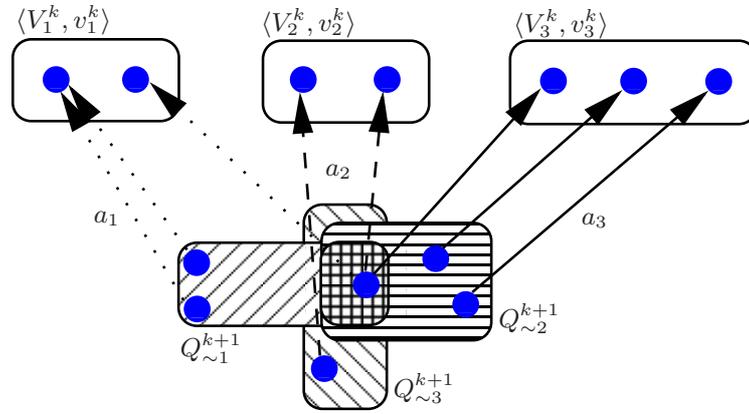


Figure 6.3: Set-based combination of partial block pre-images. The partial block pre-images are computed by regression as $Q_{\sim i}^{k+1} = \langle T_S^{-1}(V_i^k, a_i), a_i, (\gamma \cdot p_i \cdot v_i^k) \rangle$ (for $i = 1 \dots 3$) where the pts-structure of action a is defined as $\triangleright_a = [\langle a_1, p_1 \rangle, \langle a_2, p_2 \rangle, \langle a_3, p_3 \rangle]$. The intersection of the state sets in $Q_{\sim 1}^{k+1}$, $Q_{\sim 2}^{k+1}$ and $Q_{\sim 3}^{k+1}$ contains only one state s , such that $Q_{\sim i}^{k+1}$ is extended with $\langle s, a, \sum_{i=1}^3 (\gamma \cdot p_i \cdot v_i^k) \rangle$. Note that three value parts and three action outcomes give rise to a number of other possible combinations (e.g. involving regressing V_2^k through action outcome a_3), but many of these combinations result in an empty intersection.

To compute a full block pre-image for some action a , a combination is made of partial block pre-images; one for each outcome. The combination of partial block pre-images implements the summation over probabilistic outcomes in Equation 6.2. The intersection of the state sets ensures that all states in the resulting full block pre-image have a utility value that complies with this summation. If c_i denotes the number of partial block pre-images for outcome a_i of action a , the number of combinations to be considered is $\prod_i c_i$, though most of these will not result in a non-empty intersection and thus no new full block pre-image is created.

EXAMPLE 6.1.4 ► Let $\triangleright_a = [\langle a_1, p_1 = 0.3 \rangle, \langle a_2, p_2 = 0.7 \rangle]$. Let $\gamma = 0.9$, $\hat{T}(s_1, a_1, s_4) = 1$, $\hat{T}(s_1, a_2, s_5) = 1$, $\hat{T}(s_2, a_1, s_4) = 1$, $\hat{T}(s_2, a_2, s_2) = 1$, $\hat{T}(s_3, a_1, s_3) = 1$ and $\hat{T}(s_3, a_2, s_5) = 1$. Furthermore, let $\langle \{s_4\}, 10 \rangle \in V^k$ and also $\langle \{s_5\}, 5 \rangle \in V^k$. Two partial pre-images that will be created in the regression step are $Q_{\sim 1}^{k+1} = \langle \{s_1, s_2\}, \gamma \cdot 0.3 \cdot 10 = 2.7 \rangle$ and $Q_{\sim 2}^{k+1} = \langle \{s_1, s_3\}, \gamma \cdot 0.7 \cdot 5 = 3.15 \rangle$. Combining $Q_{\sim 1}^{k+1}$ and $Q_{\sim 2}^{k+1}$ yields $\langle \{s_1, s_2\} \cap \{s_1, s_3\}, a, 2.7 + 3.15 \rangle = \langle \{s_1\}, a, 5.85 \rangle$, denoting that the Q -value of doing action a in state s_1 is 5.85.

Algorithm 15 STEP IV: **Set-based Dynamic Programming**. The lines 6–10 compute $B_{\boxplus}^a V^k$ for all actions, whereas lines 6–11 implement a full backup $B_{\boxplus}^* V^k$. It is also possible to compute $B_{\boxplus}^a V^k$ and combine (line 10) for each action subsequently.

```

1: [STEP IV] : Set-Based Dynamic Programming
2:  $V^0 = R = \{\langle V_1^0, v_1^0 \rangle, \dots, \langle V_n^0, v_n^0 \rangle\}$ 
3:  $k = 1$ 
4: repeat
5:    $Q_{\sim}^{k+1} := \emptyset$ 
6:   for each  $\langle V_i^k, v_i^k \rangle \in V^k$  do
7:     for each  $a \in A$  do
8:       for each  $\langle a_j, p_j \rangle \in \triangleright_a$  do
9:          $Q_{\sim}^{k+1} := Q_{\sim}^{k+1} \cup \{\langle \hat{T}_S^{-1}(V_i^k, a_j), a_j, (\gamma \cdot p_j \cdot v_i^k) \rangle\}$ 
10:   $Q^{k+1} := \text{combine}(Q_{\sim}^{k+1})$ 
11:   $V^{k+1} := R \oplus \max_A Q^{k+1}$ 
12:   $V^{k+1} := \text{merge}(V^{k+1})$ 
13:   $\Delta := \arg \max_{\langle \cdot, \Delta \rangle} |V^{k+1} \ominus V^k|$ 
14:   $k := k + 1$ 
15: until  $\Delta < \sigma$ 
    
```

In line 10 of Algorithm 15 all partial Q -blocks are combined into normal Q -blocks. After this step, a standard maximization operation can take place to compute a maximized V -partition in line 11. The full procedure that consists of regression, value computation, combination and maximization, forms a *decision-theoretic version of regression* (DTR). Standard regression of a state s through an action a computes those states s' from which the execution of a leads to s . In decision-theoretic regression (DTR), one computes – in addition – the utility value of s' in the face of uncertainty and a reward model. Boutilier *et al.* (1999) reviews several ideas in DTR in full detail. Region-based backpropagation through a probabilistic model was already applied in early work by Lozano-Perez *et al.* (1984) and Erdmann (1986). Note that the algorithm could be used online too, by generating traces and performing block-based updates along the path.

A Note on Policy Iteration. Policy iteration in this setting can be defined analogously to the previous step. The main difference is that one can only compute the intersection of the policy with the current value function Q^{k+1} *after* the combination step. This makes policy iteration yet more complex, because first all actions have to be considered, and all their effects combined. Still, one can intersect full block pre-images for individual actions with the current policy on a per-action basis.

6.1.5 STEP V: Intensional Dynamic Programming

The final step in our five-step description is *intensional dynamic programming* (IDP) in which we replace the set representations of STEP IV by *descriptions* in a suitable language. Even though Algorithm 15 shows how DP algorithms can be constructed that utilize the MDP's structure to perform set-based (justified) backups, they still rely on the assumption that all states are stored and accessed extensionally. We have seen in Chapter 3 that for most (real-world) problems, this is unrealistic and often unnecessary. There, we have also seen that there is a wide variety of devices for generalization purposes such as neural

networks and decision trees. In this section, because of our interest in exact algorithms and justified generalization, we focus on a specific type of representational devices; namely those that can compactly represent *sets of states*. Such descriptions enormously improve the *representational economy* of the system; for example, the description *all Dutch people* is much shorter than the set consisting of all (about) 16 million names of those people.

We start this section with a summary of the core aspects of the preceding (set-based) algorithms. After that, we introduce a generic representation language to describe sets of states, that enables us to translate Algorithm 15 to the case in which compact representation languages are used. As in the preceding steps, we assume finite MDPs and discounted, infinite-horizon reward criteria.

6.1.5.1 DISCUSSION AND ANALYSIS OF SET-BASED DYNAMIC PROGRAMMING

In the preceding steps we have shown how DP algorithms, such as value iteration, policy iteration and related methods, can be extended to work with set-based representations. Here we reflect on the main components of set-based DP and we show that **i)** it is functionally equivalent to classical DP, and **ii)** it can be extended systematically to work with arbitrary state abstractions, other than aggregations.

The end product of STEP IV is a set-based DP algorithm in which structure is found in the form of state clusters (sets). Although aggregation is only a limited form of generalization, it does provide a starting point for the use of more general *languages* for state abstraction. In various other methods, generalization is often a mix of feature (region) learning and value function learning (see also Chapter 3). In contrast, Algorithm 15 employs an aggregation method that can be characterized as *justified generalization* (see for example Ellman, 1989), i.e. states are only aggregated when it is *known* that they should be aggregated. Reasons for aggregation are that states have the exact same value, or they share the same optimal policy action. This knowledge is directly derived from the MDP's model. Aggregation techniques have been used in DP contexts before, but mostly for *approximate* DP (e.g. Bertsekas and Castañon, 1989; Baum and Nicholson, 1998; Boutilier *et al.*, 1999; Zhang and Baras, 2001; Lambert III *et al.*, 2004, and see further in Chapter 3). On the contrary, we focus on *exact* algorithms where the sole purpose of aggregation is to cluster states that have *exactly* the same properties. A key point in understanding Algorithm 15 is the following principle:

If the current value function V^k is set-based, then Algorithm 15 will compute a set-based value function V^{k+1} as a refinement of V^k . The structure of V^{k+1} is only refined (in comparison with V^k) in areas where the one-step effects of actions differ with respect to the values in V^k .

More generally, the *a priori structure* that is present in set-based specifications of the reward function and probabilistic action dynamics, is utilized for computing the *a posteriori structure* of the set-based, optimal value function and policies. Both types of structure are expressed in terms of *piecewise constant*¹⁵, set-based representations, though in the following sections we will extend this principle beyond that. In the current setting, we focus

¹⁵In contrast, when discussing continuous MDPs and POMDPs, often so-called *piecewise linear* functions are used. A function f over a region \square is piecewise linear and convex, if there exists a finite set of linear functions $L = \{l_i \mid l_i(x) = A_i x + B_i\}$ such that $\forall x \in \square, f(x) = \max_{l_i \in L} l_i(x)$.

on how set-based DP is performed, and not so much on practical aspects such as data structures and computational complexity.

Algorithm 15 implements the Bellman equation for all states $s \in S$, i.e. it computes Equation 6.2. It is insightful to distinguish three core set-based operations in this process. This is useful for showing how the constituents of the Bellman equation are upgraded to a set-based context. Furthermore, it highlights the parts that must be considered when we address intensional descriptions in the next sections. One of the key mechanisms is to use *set intersections* for summation and maximization. Each iteration starts with a set-based representation of V^k and performs the following three steps:

1. **Regression:** Computing *which* states should be updated is called *regression*. For each abstract state V_i^k and deterministic outcome a_j of some action a the complete set of states is computed from which all transitions lead to a state in V_i^k . The partial value of this state can be computed from the value of the resulting state v_i^k , the probability of outcome a_j when executing action a , and the discount factor γ . Summarizing, all factors $\langle s, a, T(s, a, s') \cdot \gamma \cdot V^k(s') \rangle$ are replaced by $\langle \hat{T}_S^{-1}(V_i^k, a_j), a_j, (\gamma \cdot p_j \cdot v_i^k) \rangle$
2. **Combination:** This step combines all the effects of a probabilistic action into a set-based Q -function. The regression step delivers a number of state sets with partial Q -values that are all based on individual action outcomes. The summation in Equation 6.2 over partial Q -values for one individual state is replaced by the intersection (and summation) of whole state sets of which the members share the same partial Q -values.
3. **Maximization:** Maximization in the classical formulation assigns to each state $s \in S$ as state value $V(s)$ the highest Q -value $Q(s, a)$, over all actions $a \in A$. In the set-based formulation this is replaced by assigning to all possible intersections of the set-based Q -function the highest value of all intersecting sets.

After these three steps (or before maximization in case of state-action value functions), the resulting set-based state value function is intersected with the reward function partition, to complete the update. Together, these steps provide the basis for *set-based successive approximation* (SBSA), i.e. the iterative computation of the series of set-based value functions $V^0, V^1, V^2, \dots, V^*$. Note that classical DP, based on individual state representations, is merely a special case of the general set-based methodology (see also Proposition 6.1.2). The key point is that the choice for a compact representation (e.g. state sets) comes with an intersection operation, and together they provide the essential ingredients to construct compact regression, combination and maximization operators. In the case of action rewards (or additional action costs) the addition of rewards is included in (or done right after) the combination step *before*¹⁶ the maximization step.

Because regression, combination and maximization provide set-based replacements for the state-based operations in the optimal Bellman backup in Equation 6.2, they can be used to construct a closed-form, set-based version of that equation. Such a backup operator takes a set-based representation of V^k and computes a set-based value function V^{k+1} based on a one-step lookahead and V^k .

¹⁶This is because in that case they are important in the maximization process over actions.

DEFINITION 6.1.14 ▶ Let $M = \langle S, A, T, R \rangle$ be an MDP and let $V^k = \{\langle V_1^k, v_1^k \rangle, \dots, \langle V_n^k, v_n^k \rangle\}$ be a set-based value function over S .

The **set-based Bellman backup operator** \mathbf{B}_{\boxplus}^* is defined as follows:

$$\begin{aligned} V^{k+1} &= \left(\mathbf{B}_{\boxplus}^* V^k \right) \\ &= \text{merge} \left(R \oplus \max_A \left(\underbrace{\text{combine} \left(\bigcup_{i,a,j} \left\{ \underbrace{\langle \hat{T}_S^{-1}(V_i^k, a_j), a_j, (\gamma \cdot p_j \cdot v_i^k) \rangle}_{\text{regression}} \right\}}_{\mathbf{B}_{\boxplus}^a, \forall a \in A} \right) \right) \right) \end{aligned} \quad (6.5)$$

For set-based value functions, \mathbf{B}_{\boxplus}^* thus replaces \mathbf{B}^* . Furthermore, \mathbf{B}_{\boxplus}^a is the set-based version of \mathbf{B}^a and is highlighted in the equation above. The backup operator \mathbf{B}^π is replaced by $\mathbf{B}_{\boxplus}^\pi$ that intersects the outcome of \mathbf{B}_{\boxplus}^a with a set-based policy.

This closed form of set-based Bellman backups combines structural operations with parameter computations. The *structural part* consists of the state set arrangement that is constructed in the backup. The *parameter part* consists of the values that are computed for all state (-action) sets. The fact that both parts are computed simultaneously places the computation at PIAGET-3 level. And although we focus on backup operators, Equation 6.5 can also be interpreted as a Bellman equation in the same way as in the classical framework (see Chapter 2). Related Bellman equations based on state aggregations and individual state transitions can be found in both model-free (Singh *et al.*, 1995) and model-based (Givan *et al.*, 2003) contexts. The conceptual idea is summarized nicely in the following quote.

Dietterich and Flann (1995, p. 3): *”Explanation-based learning provides an alternative approach to state generalization. The goal regression step of EBL is very closely related to the Bellman backup step of RL. A Bellman backup propagates information about the value of a state backwards through an operator to infer the value of another state. Goal regression propagates information about the value of a set of states backwards through an operator to infer the value of another set of states.”*

One thing not present in classical DP algorithms is the use of a merge operation. This operation is needed to keep the set-based representation as coarse as possible. Without it, the value function may be shattered much more than needed. From a more general point of view, the merge-operation represents a first kind of KR efforts. Because of the nature of the algorithms we have introduced, it is important to keep representations compact to reduce the amount of work when computing the next iteration, i.e. the complexity of the backup is proportional to the size of the current V^k -partition.

Despite the operational differences between classical and set-based VI (Algorithms 12 and 15), we can show that they compute the exact same values, i.e. the backup operator \mathbf{B}_{\boxplus}^* implements \mathbf{B}^* , something that cannot come as a surprise given that \mathbf{B}_{\boxplus}^* was built from the classical Bellman backup operator by abstracting over operations. First, let V be a value function for a state space S . A set-based representation V_S derived from V is constructed using $V_S := \text{merge}(\{\langle s, v \rangle \mid s \in S, V(s) = v\})$.

PROPOSITION 6.1.2 ▶ Let $M = \langle S, A, T, R \rangle$ be an MDP, let V^k be a value function over S and let V_S^k be a set-based representation derived from V^k . For all states $s \in S$ it holds that

$$V^{k+1}(s) = \left(\mathbf{B}^* V^k \right)(s) = \left(\mathbf{B}_{\boxplus}^* V_S^k \right)(s) \quad (6.6)$$

Proof. By construction. In the preceding steps, we have described how each step is functionally equivalent to its predecessor. Together, this makes Algorithm 15 equivalent to Algorithm 12. \square

With this result it is easy to see that set-based DP converges to a value function that is optimal at the individual state level.

PROPOSITION 6.1.3 ▶ The **optimal value function** $V^* = \{\langle V_1^*, v_1^* \rangle, \dots, \langle V_n^*, v_n^* \rangle\}$ satisfies the following **fixed point** definition:

$$V^* = \left(\mathbf{B}_{\boxplus}^* V^* \right) \quad (6.7)$$

This follows directly from Proposition 6.1.2 and it shows that Algorithm 15 computes exactly the same value functions $V^0, V^1, \dots, V^k, V^{k+1}, \dots, V^*$ as classical DP, but multiple updates are performed simultaneously by making use of the MDP's structure.

The computational complexity of set-based DP is an important issue. The complexity class, i.e. the worst-case complexity of DP algorithms (e.g. see Littman *et al.*, 1995; Mansour and Singh, 1999, and further in Chapter 2) is polynomial in the size of the state space and this is left unchanged in Algorithm 15. The operations (merge, max, combine and regression) in Equation 6.5 all have polynomial complexity. Let ρ be the average number of states covered by each of the abstract states in the value function V^k . The number of full backups is decreased from $|S|$ to $\frac{|S|}{\rho}$, though each backup in set-based DP consists of several set-based manipulations that make the algorithm work on $|S|$ states after all. This will change in STEP V when state sets are described rather than explicitly represented.

In contrast to the complexity of Algorithm 15, its convergence properties are slightly different from those of classical DP algorithms. The distinction between a structural and a parameter part in the set-based Bellman backup can also be carried over to the convergence of set-based DP. Structural convergence means that the aggregations become stable, i.e. the states within an aggregate are equivalent with regard to arbitrarily longer horizon lengths. Parameter convergence amounts to the more classical meaning of convergence of values in DP algorithms.

Set-Based Policy Iteration and Other Variations. Policy iteration consists of an *evaluation step* and an *improvement step*. Evaluating a policy can largely be done along the lines of Algorithm 15, except that after a regression operation, the resulting state set should be intersected with the set-based policy π as we explained in Section 6.1.3, effectively implementing $\mathbf{B}_{\boxplus}^\pi$. In this way, the algorithm computes V^π .

Improving the policy can most easily be performed by first replacing the maximization operation by the more specialized *max* operation over Q -functions (see Definition 6.1.9) to obtain Q_{\max}^{k+1} and then deriving the *greedy* policy π^{k+1} from that by discarding the action values. Note that in our description, we emphasize the structural part of algorithms where

a policy is used during regression, and extracted from a value function. Another possibility is to *sample* a structured policy in an interaction process with the environment (e.g. see Fern *et al.*, 2006).

The order of the operations in Equation 6.1.2 (and Algorithm 15) has been given in such a way that the original Bellman equation is clearly recognizable. However, the equation expresses *what* must be computed, but other sequences of operations that compute the same value sets are possible. For example, in Section 6.3 we use a *sequential* combining operator that combines the previous first outcome of an action with the second, after which the result is combined with possibly a third and so on. In this way, the order of operations is altered because of efficiency reasons. However, as with classical DP algorithms, many variations on either the order, or the selection, of updates are possible (see Section 2.5.2). In this case, various *asynchronous* or *modified* IDP algorithms can be formed. There are many opportunities for further research here, but some existing examples are the structured prioritized sweeping algorithm by Dearden (2001), or the search-based, first-order variation of IDP, FOVIA (Karabaev and Skvortsova, 2005)

Other extensions and variations can be employed in multi-agent decision making, for example in the form of *Markov games* are straightforward extensions of IDP. For example, Finzi and Lukasiewicz (2004c) use intersections of two agent's value partitions as a multi-agent extension of the SDP framework introduced by Boutilier *et al.* (2001)

Approximate Methods. Exact representations of set-based value functions only aggregate states that have exactly the same value (or optimal policy action). If there are many distinct values, the number of abstract states may grow large, which affects the computational complexity of a set-based DP algorithm. A remedy to this problem is to apply looser criteria for aggregation in the merge-function, for example to aggregate states when their values differ at most some threshold. This creates larger abstract states, but does affect the precision of the values (i.e. values are averaged throughout the abstract state, or they can be represented as intervals).

A second opportunity for approximation lies in the *structural*¹⁷ aspects of Algorithm 15. The algorithm halts when the maximum distance Δ between two subsequent value functions V^k and V^{k+1} is small enough. After each iteration, V^{k+1} is a state space partitioning in which states are aggregated when they have the same properties relative to the length of the horizon computed so far. Instead of iterating another time, one can fix the abstract state space spanned by V^{k+1} and use classical model-free or indirect learning over this space to estimate a state value function. Value learning on this abstract space is computationally cheaper than refining the partition through Algorithm 15, though it must be ensured that the partition is fine enough.

6.1.5.2 INTENSIONAL DESCRIPTIONS OF STATES

Extensional representations of sets of states can be replaced by *intensional*¹⁸ *descriptions* by making use of a suitable KR framework. Similar to the case of FOL (see Section 4.2.1), we want to have a *syntactic* (see Section 4.2.1.1) formalism to *describe*, in a compact and

¹⁷For example, Boutilier *et al.* (2000a) discuss stopping their SPI algorithm at some point when trees become too large.

¹⁸Intensional here means the counterpart of extensional. And although there some connections with *S5* modal logic, we do not go further into this here.

efficient way, structures that form the *semantic* (see Section 4.2.1.2) layer of the system, i.e. the sets we have used up to here. The main requirement is that the semantics is formed by an MDP's state space and that the language supplies ways to express *properties* that are shared throughout subsets of the state space¹⁹. In that sense, states are equivalent to *interpretations*, and the language should make use of similarities between states, or structure within a state's description itself.

In the following, we introduce the minimal requirements of so-called *state description languages* needed to construct an intensional version of Algorithm 15. Such languages can be based on propositional or first-order logic, and later in this section we describe three examples in the context of MDPs.

DEFINITION 6.1.15 ▶ Let $M = \langle S, A, T, R \rangle$ be an MDP. A **state description language** \mathbb{L} over S is any formal system that can be used to describe properties of states in S . Elements of \mathbb{L} are called **abstract state descriptions** or simply **abstract states**. The set S provides a semantics for the symbols in \mathbb{L} . The **denotation** of $X \in \mathbb{L}$, denoted $\llbracket X \rrbracket$, is a subset of S . For all $s \in S$ we assume that $s \in \mathbb{L}$, and that $\llbracket s \rrbracket = \{s\}$, resembling Herbrand semantics (see Section 4.2.2.1). When $s \in \llbracket Z \rrbracket$ for some $Z \in \mathbb{L}$, then Z is said to **cover** s .

Reasoning on the syntactic level is formalized as the generic proof relation $\vdash_{\mathbb{L}}$. Let $X_1, X_2 \in \mathbb{L}$, then $X_1 \vdash_{\mathbb{L}} X_2$ iff²⁰ $\llbracket X_1 \rrbracket \subseteq \llbracket X_2 \rrbracket$. Consequently, a state $s \in S$ is covered by an abstract state X iff $s \vdash_{\mathbb{L}} X$, and two expressions $X_1, X_2 \in \mathbb{L}$ are called equivalent, denoted $X_1 \equiv_{\mathbb{L}} X_2$ iff $\llbracket X_1 \rrbracket = \llbracket X_2 \rrbracket$.

Thus, a state description language \mathbb{L} contains expressions that describe subsets of a state space S . There are no other restrictions on the language, and background knowledge nor complete domain theories are excluded. Any such language typically contains *operators* or *connectives* that can be used to form complex expressions from simple constituents. Semantically this amounts to operations on the state sets covered by these expressions.

DEFINITION 6.1.16 ▶ Let \mathbb{L} be a state description language over S . We assume that \mathbb{L} is productive and closed off under a set of operators (or, *logical connectives*, see Chapter 4) that allow for building more complex descriptions (i.e. abstract states). Let $X_1, X_2 \in \mathbb{L}$ be two abstract states. A syntactic operator $\circ_{\mathbb{L}}$ yields²¹ $Y \equiv X_1 \circ_{\mathbb{L}} X_2$ where Y is a new expression in \mathbb{L} . Each operator $\circ_{\mathbb{L}}$ comes with a semantic counterpart $\bullet_{\mathbb{L}}$ that is defined such that $\llbracket Y \rrbracket = \llbracket X_1 \rrbracket \bullet_{\mathbb{L}} \llbracket X_2 \rrbracket$ whenever $Y \equiv X_1 \circ_{\mathbb{L}} X_2$.

At the very least, we assume that \mathbb{L} comes along with a semantic **intersection** and a **difference** operator: let $X_1, X_2 \in \mathbb{L}$, then the intersection Y of X_1 and X_2 is defined as $Y \equiv X_1 \wedge_{\mathbb{L}} X_2$, $Y \in \mathbb{L}$, and $\llbracket Y \rrbracket = \llbracket X_1 \rrbracket \cap \llbracket X_2 \rrbracket$, and the difference Z is defined as $Z \equiv X_1 \text{sdif}_{\mathbb{L}} X_2$, $Z \in \mathbb{L}$ where $\llbracket Z \rrbracket = \llbracket X_1 \rrbracket - \llbracket X_2 \rrbracket$.

Most languages will support operators for the intersection and union of state sets. The

¹⁹In the propositional setting, we focus on state abstraction. Action abstraction was considered by only a few systems and requires that a description language must also provide means to describe properties of actions. While we omit this aspect here, it comes naturally in the relational setting in the second half of this chapter where we deal extensively with action abstraction too.

²⁰This implies that we assume soundness and completeness of the proof system generated by $\vdash_{\mathbb{L}}$.

²¹Logical connectives such as \wedge and \vee maintain the syntax of the constituents, i.e. $\varphi \wedge \psi$ simply is $\varphi \wedge \psi$. The more general *operator* keeps open the possibility that when $\chi \equiv \varphi \wedge \psi$, the exact syntactic form of the expression χ can be anything (within the language).

expressivity of \mathbb{L} determines which $S' \subseteq S$ can exactly be characterized by a single $\mathbb{X} \in \mathbb{L}$. Most languages over some set S can only represent some (type of) subsets of S . For example, propositional tree representations (e.g. see Section 3.5) cannot directly represent disjunctive formulas, and 3-dimensional hyperplanes in a continuous space (i.e. cubes) cannot directly represent a conus-shaped subspace. The *tractability* of \mathbb{L} is about how efficiently $\vdash_{\mathbb{L}}$ can be implemented and furthermore how much computation is involved when computing with expressions (e.g. implicitly intersecting the underlying state sets). The trade-off between expressivity and tractability typically makes that simpler languages, which have better computational properties, are favored (see Brachman and Levesque, 2004, for more on these matters and also Chapter 4).

Although we do not discuss any specific language in this section, we do assume that most languages will not be expressive enough to describe *any* set using just one language expression. Thus, value functions, reward functions and policies will generally hold more descriptions than there are sets with distinct values or actions. And furthermore, although we describe our algorithm in terms of a generic language with operators, most advanced or practical systems will use more efficient representations and data structures (trees, ADDs, BDDs, etc). We have seen examples of such representations in FOL, in Section 4.2.3.

DEFINITION 6.1.17 ▶ The abstract state descriptions in a state description language \mathbb{L} are partially ordered by a generic relation $\preceq_{\mathbb{L}}$. Let \mathbb{X} and \mathbb{Y} be two abstract states in \mathbb{L} . If $\mathbb{Y} \preceq_{\mathbb{L}} \mathbb{X}$, then \mathbb{X} is said to be **more general than** \mathbb{Y} (and \mathbb{Y} is **more specific than** \mathbb{X}), and $\llbracket \mathbb{Y} \rrbracket \subseteq \llbracket \mathbb{X} \rrbracket$. The **reduced form** of an expression $\mathbb{Y} \in \mathbb{L}$ is the smallest²² expression \mathbb{Y}' for which $\mathbb{Y} \preceq_{\mathbb{L}} \mathbb{Y}'$ and $\mathbb{Y}' \preceq_{\mathbb{L}} \mathbb{Y}$.

Thus, the reduction of an expression is defined relative to the ordering $\preceq_{\mathbb{L}}$ though other, equivalent expressions might be found as well. The purpose of reductions is to find smaller descriptions that cover the exact same state set. For some languages, generality coincides with the accompanying (syntactic) proof relation $\vdash_{\mathbb{L}}$. For example, for Datalog, θ -subsumption coincides with (natural) deduction (see Section 4.3.2.2). In general, it is convenient to separate these two matters. Furthermore, the ordering can be used to *compare* descriptions and possibly discard descriptions that are covered by more general ones, or as a search space to find more specific or more general descriptions, as in ILP (see Section 4.3.2).

DEFINITION 6.1.18 ▶ Let $M = \langle S, A, T, R \rangle$ be an MDP and let \mathbb{L} be a state description language over S . A **value** (or **reward**) **function** is a set $\mathbb{V} = \{\langle \mathbb{V}_1, v_1 \rangle, \dots, \langle \mathbb{V}_n, v_n \rangle\}$, where each \mathbb{V}_i is an abstract state, v_i is a real value ($i = 1 \dots n$), and $\mathbb{V}_1, \dots, \mathbb{V}_n$ form a covering²³ of S . For any state $s \in S$,

$$\mathbb{V}(s) = \max\{r_i \mid \langle \mathbb{V}_i, r_i \rangle \in \mathbb{V} \wedge s \in \llbracket \mathbb{V}_i \rrbracket\}$$

A **policy** is a set $\mathbb{P} = \{\langle \mathbb{P}_1, a_1 \rangle, \dots, \langle \mathbb{P}_n, a_n \rangle\}$, where each \mathbb{P}_i is an abstract state, $a_i \in A$ is an action ($i = 1 \dots n$), and $\mathbb{P}_1, \dots, \mathbb{P}_n$ form a covering of S . For any state $s \in S$, $\mathbb{P}(s) = a \Leftrightarrow a \in \{a_i \mid \langle \mathbb{P}_i, a_i \rangle \in \mathbb{P} \wedge s \in \llbracket \mathbb{P}_i \rrbracket\}$.

The semantics of value functions is taken more general than in the previous steps. Because

²²Although 'smallest' can mean many things here, a measure of the size of the expressions suffices.

²³We can ensure this by defining default rules.

of a possibly limited expressivity of the language, we assume that a value function can contain multiple abstract states that may overlap. To uniquely define the value of state, taking the maximum of all covering states suffices. Value and reward functions are assumed to be *piecewise constant*, which means that all states covered by an abstract state share the same value. The plus (\boxplus) and minus (\boxminus) operations on value functions in Algorithm 16 are direct translations of the (semantic) set-based plus (\oplus) and minus (\ominus), by substituting set intersections with (syntactic) overlap computations (see also Definition 6.1.16).

Actions can be formalized in a state description language as follows:

DEFINITION 6.1.19 ▶ A **probabilistic action** a is a tuple

$$\left\langle \mathbb{C}, a, \left[\langle \mathbb{E}_1, p_1 \rangle, \dots, \langle \mathbb{E}_n, p_n \rangle \right] \right\rangle$$

where \mathbb{C} are the **preconditions**, \mathbb{E}_i ($i = 1 \dots n$) are **outcomes**, $0 \leq p_i \leq 1$ and $\sum_{i=1}^n p_i = 1$. The **procedural semantics** of a is as follows: if the current state is described as \mathbb{S} , and $\mathbb{S} \vdash_{\perp} \mathbb{C}$, then the resulting state after executing a in state \mathbb{S} , is described by \mathbb{S}' with probability p_i ($i = 1 \dots n$). The syntactic form of \mathbb{S}' is computed²⁴ from \mathbb{S} and \mathbb{E}_i .

A probabilistic action a induces a transition function over S in the following way: if the current state $s \in \llbracket \mathbb{S} \rrbracket$ then with probability p_i ($i = 1 \dots n$) the resulting state after executing a in s is $s' \in \llbracket \mathbb{S}' \rrbracket$.

The standard semantics of such an action is given by the fact that it transforms the syntactic form of the pre-state into the syntactic form of the post-state. Each such description covers a set of states, rendering the standard semantics of probabilistic actions a standard probabilistic transition function. Note that this definition bears similarities to that in Chapter 4 (see Definition 4.5.3) for probabilistic actions in first-order domains. An important difference is that here we aim at generic patterns, regardless of the syntax of the language, the specifics of the operational semantics, or the underlying semantics (i.e. the exact nature of the states). In fact, most action formalisms need additional operational semantics to determine how state descriptions change as an effect of actions. An example is the union (and subtraction) of the ADD (and DEL) lists in the STRIPS action formalism (see Section 4.4.2.1).

A major difference with the set-based formalisms we have used for Algorithms 12 to 15 is that action transition probabilities are not specified for individual transitions, but instead for entire sets of transitions. In other words, in Definition 6.1.19 PTS-uniformity is built-in *because* it is defined for multiple individual transitions simultaneously.

DEFINITION 6.1.20 ▶ An MDP **based on** the state description language \mathbb{L} consists of a set of states S , a set of probabilistic actions A as defined in Definition 6.1.19 and a reward function defined as in Definition 6.1.18.

This definition has clear connections with FORMs (see Definition 4.5.5 in Chapter 4). We will discuss examples of instantiations of state description languages and MDPs based on them in Section 6.1.5.4.

²⁴For ease of explanation in the remainder of this chapter, we will assume that each effect \mathbb{E}_i specifies the complete post-state description, see also our relational language in the next sections. However, for some action formalisms this will not be the case, and additional efforts are needed, as well as some additional notation. However, the general setup of IDP is not changed by this.

6.1.5.3 INTENSIONAL DYNAMIC PROGRAMMING

IDP combines algorithmic procedures of set-based DP with representational aspects of state description languages. This enables **i)** elegant and compact specifications of huge sequential decision making problems in stochastic domains, and **ii)** efficient computation of optimal policies for these problems *without explicit state space enumeration*. Therefore, the core of IDP can now be formulated as follows:

$$\text{set-based DP} + \text{knowledge representation} = \text{intensional DP}$$

Any choice for a particular KR formalism to describe states determines²⁵ most representational aspects (including opportunities for efficient data structures) and algorithmic aspects (such as operations on abstract states). Expressivity of the language directly influences the complexity of IDP, because it determines how precisely specific state sets can be described by elements in the state description language. Quoting Givan *et al.* (2003, p165) in the related context of model minimization: *”This ... leads to an interesting trade-off between the strength of the representation used to define the aggregates (affecting the size of the reduced MDP), and the cost of manipulation operations. Weak representations lead to poor model reduction, but expressive representations lead to expensive operations.”* A key insight into IDP is the following rationale behind Algorithm 16:

If the current value function \mathbb{V}^k is described in language \mathbb{L} , then Algorithm 16 will compute a value function \mathbb{V}^{k+1} in language \mathbb{L} as a refinement of \mathbb{V}^k . The structure of \mathbb{V}^{k+1} is only refined (in comparison with \mathbb{V}^k) in areas where the one-step effects of actions differ with respect to the values in \mathbb{V}^k .

In the following paragraphs we define intensional counterparts of the main three operations in set-based DP, which are *regression*, *combination* and *maximization*. In addition, because the transition function is no longer state-based, we introduce *overlap* as a fourth operation and because IDP works with descriptions a fifth type of operation, *simplification* is introduced. All five operations (OP1–OP5) work at the syntactic level, having the set-based operations as their semantics.

OP1: Overlaps. In the previous steps we have assumed that T is defined state-wise, such that pre-images can be computed for any state set S' by just looking at the states in S' . With the introduction of abstract actions in Definition 6.1.19 this is no longer possible. The purpose of *intersection* (or *matching*) is to find a description of an abstract state that describes exactly those states that are covered by an abstract state in the value function *and* the effects of a (probabilistic) action.

DEFINITION 6.1.21 ► The **overlap** of two abstract states in a language \mathbb{L} over S is a function of type $\mathbb{L} \times \mathbb{L} \rightarrow \mathbb{L}$. Let $\mathbb{X}, \mathbb{Y} \in \mathbb{L}$, then $\text{overlap}(\mathbb{X}, \mathbb{Y}) =_{\text{def}} \text{reduce}(\mathbb{X} \wedge_{\mathbb{L}} \mathbb{Y})$.

This means that the semantics of the overlap is given by the intersection of state sets. That is, let $\mathbb{X}, \mathbb{Y} \in \mathbb{L}$, then if $\mathbb{Z} = \text{overlap}(\mathbb{X}, \mathbb{Y})$ then $\llbracket \mathbb{Z} \rrbracket = \llbracket \mathbb{X} \rrbracket \cap \llbracket \mathbb{Y} \rrbracket$. The overlap in Figure 6.3a is used to compute a description of where an abstract state \mathbb{V}_1^k in the value function and a deterministic outcome \mathbb{E}_j of an action coincide.

²⁵It seems like finding a syntax (e.g. a language) for the semantics (i.e. set-based DP) we already have.

OP2: Regression and Weakest Preconditions. Algorithm 15 uses the inverse \hat{T}_S^{-1} of the transition function to find the set of states that can reach a subset in the value function. Its *descriptive* counterpart is a technique generally known as *regression* and the block pre-image counterpart will be called *weakest precondition* (wp). Regression from a description \mathbb{V}_i in the current value partition (i.e. a set of states) is used to find a description of a block pre-image. In other words, the value region \mathbb{V}_i is *pushed through* the action a to get abstract state descriptions along with their decision-theoretic values. Regression is purely syntactic, but its semantics mimic computing set inverses such as in Algorithm 15.

DEFINITION 6.1.22 ▶ Let $\langle \mathbb{C}, a, [\langle E_1, p_1 \rangle, \dots, \langle E_n, p_n \rangle] \rangle$ be a probabilistic action, and let \mathbb{S} be an abstract state in a state description language \mathbb{L} over a state space S .

The i th **weakest precondition** of \mathbb{S} **given action** a , denoted $\text{wp}_i(\mathbb{S}, a)$, is the most general abstract state description \mathbb{S}' such that executing action a in a state s' that is covered by \mathbb{S}' , will lead to a state s covered by \mathbb{S} , with probability p_i .

Thus, semantically, regression computes precisely the inverse of actions. The challenge, however, is to outline how the syntactic shape of the states *before* the action can be computed from the description of the states *after* applying the action. That is because standard action definitions (such as Definition 6.1.19) prescribe only how to compute the syntactic form of the state description *after* an action from the description of what holds *before* the action is executed. Depending on the action formalism or logic, regression may be axiomatized.

Figure 6.4a²⁶ gives an example of regression. First, the overlap between an action outcome E_j and an abstract state in the value function, \mathbb{V}_1^k is computed. The regression step computes a description of all the states that can reach a state in the overlap, if the particular outcome E_j of the action occurs. The resulting description $\text{wp}_k(\text{overlap}(E_j, \mathbb{V}_1^k), a)$ is a refinement of \mathbb{C} (the preconditions of action a) because it must be ensured that the action is applicable throughout this state set.

Additional Notes on Regression and Weakest Preconditions. Intuitively regression is a *backwards* reasoning process of which the end-product is called the *weakest precondition*. Here, backward means that the *inverse* of a normally *default* direction is taken, for example the inverse of an action definition, or the inverse of a deduction rule. In the more limited scope of upgrading set-based DP to IDP, regression²⁷ can be understood simply as a technique that computes set inverses at language-level, i.e. reasoning from post-action to pre-action descriptions of states. Nevertheless, regression has a much wider scope and there have been many influential ideas and techniques on which it is based. Some of these are directly relevant to relational RL in general, and the techniques in this chapter in particular. Here we briefly discuss a number of them, both because of historical relevance, as for providing conceptual context.

Some of the most original ideas on regression stem from *planning* and *program verification*. Many action formalisms support the use of an action definition as an axiom

²⁶Other graphical representations of structured Bellman backups can be found in (Dean *et al.*, 1998, Appendix A), (Dietterich and Flann, 1997) and (Feng *et al.*, 2004).

²⁷Depending on your background, regression may be associated with what we would call *numerical regression*, i.e. fitting a real-valued function on data. Methods doing exactly that in the context of MDPs are generally referred to as *value-function approximation* (VFA) methods, which are discussed mainly in Chapters 3 and 5.

or inference rule, such that *deduction* can be used to reason from pre-state to post-state. The *inverse* of such axioms or rules can be used to deduce the effects of regression, but this often makes regression *unsound*, and additional reasoning mechanisms (e.g. a domain theory, see later in this chapter) are needed to make it work. Regression planning is usually performed starting from some (declarative) goals, and working backwards, transforming goals into subgoals along the way. Translated to the context of MDPs we have seen that this amounts to transforming²⁸ n -step value functions into $(n + 1)$ -step value functions.

Regression can be performed using explicit regression rules, or it can be computed by passing back information through the defining relations if a goal relation is defined in terms of other relations. In the same way, if a plan step is defined in terms of simpler component steps, then knowing how to pass relations back over components allows one to pass the relation back over the original plan step (see also hierarchical RL in Section 3.8). Yet, if no more information is available, one can make the assumption that a plan step has absolutely no effect on the relation, resembling what is usually done in forward reasoning in the *frame problem* (see Section 4.4). In all cases, regression is very much representation-dependent, and so is its complexity.

Waldinger (1975, see also (Manna and Waldinger, 1978; Waldinger, 1977)) is probably the first to discuss regression in AI, linking it to planning via *plan modification*. In order to achieve goals P and Q , construct a plan F that achieves P , and then modify F so that it achieves Q while still achieving P . The idea is to *protect*²⁹ P so that the choice of where to place the steps for achieving Q is determined relative to the plan for P . Waldinger's regression is based on the idea of *weakest preconditions* proposed by Dijkstra (1975) in the area of program verification. Manna and Waldinger (1978) use it in a theorem-proving approach to program *synthesis*. Following Waldinger, Nilsson (1980) discusses regression in the context of the STRIPS formalism (see Section 4.4.2.1), including partially grounded actions and the interaction with a domain theory. Nilsson constructs so-called *B-rules*, which are the inverse of the standard action definitions (the so-called *F-rules*) and provides a regression operator. It is this work that is most closely related to REBEL in this chapter.

Another early effort in formulating regression over simple (non-sensing) actions is due to Pednault (1989, and see also his PhD thesis) who formulated sound and complete regression operators in the ADL formalism (see Section 4.4.2.1) that extends STRIPS and allows, amongst other things, conditional effects. Addressing similar problems, Reiter (see 2001, for an overview) also presents a sound and complete formula-based regression formulation over simple actions within the situation calculus (see also Section 4.4.2.2 where we have introduced $\text{REG}(\varphi)$). It reduces reasoning about future situations to reasoning about the initial situation using first-order theorem proving. Regression operators are provided for the formulae, with and without functional fluents. Many other issues concerning goal regression and regression planning are often discussed in the general context of action languages and planning (e.g. Reiter, 2001; Russell and Norvig, 2003) and many of

²⁸One can also see this as model minimization by refining the value partition (Givan *et al.*, 2003). The key to understanding regression as minimization lies in thinking of the subgoals generated by regression as representing sets of states (those states that satisfy the subgoal). Each such set of states shares a simple property: it is the set of states that can reach the goal under a particular action sequence, and so on.

²⁹A nice illustration of problems with the conflicts between subgoals, and non-interleaving planning, is the *Sussman anomaly* that can easily trick simple blocks world planners (e.g. see Russell and Norvig, 2003, p. 410 and 414).

them are relational and particularly relevant for this book.

A second area that relates to regression and weakest preconditions is *explanation-based learning* (EBL) (Ellman, 1989; Minton *et al.*, 1989), which Mitchell (1997) classifies as *analytical learning*, contrasting it with *inductive* learning methods such as ILP (see Section 4.3.2). Inductive learning takes as input an incomplete domain theory and a set of examples and produces a richer domain theory that explains the classification of the examples. In contrast, EBL employs *deductive* proofs (i.e. an *explanation*), and uses these to generate *generalized explanations* that can be applied to similar, yet different, examples. Furthermore, the generalized explanation is operationalized, in the sense that it will only contain predicates used in the description of the example. In other words, EBL extracts general rules from individual observations³⁰. In state-space problems and MDPs, proofs correspond to showing that a sequence of actions achieves a goal, and EBL corresponds to goal regression over an operator sequence. Unlike inductive approaches to state generalization, EBL chooses regions based on the states where a specific operator (or a sequence of operators) is applicable. As with EBL in concept learning, this provides a form of *justified generalization* over states. EBL does not need to gather experience over a region and then make an inductive leap. Indeed, EBL is at the heart of the generalization algorithms used in classical systems such as PRODIGY and SOAR to learn general control rules from specific examples of problem solving episodes (see Mitchell, 1997, Section 11.4), though recently also (inductive) relational RL has been used (e.g. see Nason and Laird, 2004a).

EBL assumes a correct and complete operator model, and employs *justified generalization* using this model. Essentially, one can say that EBL is an *example-guided reformulation of theories*, or more simpler, EBL merely *restates what is already known to the learner*. This makes EBL a *knowledge compilation* or *knowledge transformation* method, or more general a *speed-up learning* method. These methods speed-up a brute-force problem solver by learning appropriate control knowledge, i.e. learning what to do, and when. A much related technique in logic programming is *partial evaluation* (PE). This is a simple knowledge compilation method that produces logically redundant rules to allow similar problems to be solved more efficiently. Both EBL and PE transform a given domain theory into a format that is more applicable to a specific situation and may have a more efficient operational format. Cast into the MDP framework, knowledge compilation (or, *program transformation*) is the compilation of value functions and policies from the domain description (i.e. an MDP). This is exactly what the IDP framework does: it transforms the model description into a description of the value function (and policy). All this knowledge can be considered a deductive consequence of the model, though IDP transforms it into a more suitable (and operational) form. EBL and PE are essentially equivalent, as has been proved by van Harmelen and Bundy (1988).

Regression, speedup learning, EBL and PE have not much been used in probabilistic domains. On the other hand, a relevant aspect for this book is that most of them started out using relational representations. Our IDP formulation uses regression over a set of deterministic transitions, and uses a language-level *combination* procedure to cope with probabilistic effects. There are several other related mechanisms in the literature for specific representations, e.g. such as *abductive repartitioning* (Boutilier *et al.*, 2000a) and *probabilistic Horn abduction* (Poole, 1997b). Furthermore, Boutilier *et al.* (2000a) dis-

³⁰EBL has its roots in STRIPS planning: when a plan was generated, a generalized version of it was saved in a plan library and later used as a macro-operator (Russell and Norvig, 2003, p. 708).

tinguish between *stochastic regression* (e.g. EBRL, Dietterich and Flann, 1997) which is regression of goals in probabilistic domains and *decision-theoretic regression* which is regression of general reward functions in probabilistic domains. Recently, De Raedt *et al.* (2007a) studied *probabilistic EBL*.

Summarizing, there are many techniques that are more or less similar to regression, and some of them can even be used in probabilistic contexts. All of them use some means to reason backwards through actions or through domain rules, and most of them generalize explanations or state space paths to employ them for other, similar cases. Even without the generalization step, regression can be employed usefully in brute-force computation of evaluation functions (e.g. see Romein and Bal, 2003; Murtagh and Utgoff, 2006). Because regression is a deductive method, one can again, as in Section 4.4.2, distinguish between action *formalisms* and action *logics*. For some languages, for example STRIPS, ADL but also some other languages concerning propositional trees or hyperplanes (see later this section), new operational definitions of how weakest preconditions should be computed, are needed. For other languages, such as the situation calculus, regression is axiomatized. Lately, research in action logics has been concerned with regression with *sensing actions*, involving modal logics, POMDPs and belief states (see Reiter, 2001, for pointers to the literature).

OP3: Combination. For a language-level version of the combination procedure in Algorithm 15 (see Definition 6.1.13), all that is needed is an intersection operator $\wedge_{\mathbb{L}}$ which gives the opportunity to intersect state sets. One difference with the set-based setting is that the combination operator can now directly use the structure of the probabilistic action (see Definition 6.1.19) instead of relying on a PTS-uniform decomposition of an initial set of general, probabilistic actions.

DEFINITION 6.1.23 ▶ The **combination** of a set of partial block pre-images \mathbb{Q} in language \mathbb{L} is defined as: $\text{combine}(\mathbb{Q}) = \{ \langle \text{reduce}(\bigwedge_{i=1}^n \mathbb{S}_i), a, \sum_{i=1}^n q_i \rangle \mid \langle \mathbb{S}_1, a_1, q_1 \rangle, \dots, \langle \mathbb{S}_n, a_n, q_n \rangle \in \mathbb{Q}, \langle \mathbb{C}, a, [\langle \mathbb{E}_1, p_1 \rangle, \dots, \langle \mathbb{E}_n, p_n \rangle] \rangle \in A \}$

Another difference with the set-based setting is that presumably the combination operator is invoked more often, depending on the expressivity of the language. Figure 6.4b shows the combination of several weakest preconditions for an action consisting of three outcomes, generating two full pre-images.

OP4: Maximization. Maximization in IDP relies – in very much the same way as the combination operator – on an intersection operator $\wedge_{\mathbb{L}}$ in a language \mathbb{L} . In addition, a language-level, *difference* operator $\text{sdif}_{\mathbb{L}}$ is needed that for any two sets S_1 and S_2 computes a description of the set $\llbracket S_1 \rrbracket - \llbracket S_2 \rrbracket$. For example, the top region in Figure 6.4c with value 2 has its size decreased after the maximization step because there are other abstract states with higher values that cover the same states. In a similar way as Definition 6.1.23 is a lifted version of Definition 6.1.13, one can redefine the maximization operator in Definition 6.1.9 to an intensional context by making use of language based intersection and difference operators.

OP5: Simplification. In contrast to the set-based setting of STEP IV, intensional descriptions require efforts to keep descriptions compact. Simplification operations are, again, highly-representation dependent, but we can distinguish a number of general mechanisms

that can be instantiated in various concrete formalisms. We have already encountered the reduce-operation on individual state descriptions. For partitions and coverings, we have at least two options. The first is a simple operation to throw away useless descriptions that may have been computed during IDP and which are included in value functions. The general idea is to throw away descriptions that are *dominated* by other descriptions which can be decided using the *ordering* on descriptions we have defined in Definition 6.1.17.

DEFINITION 6.1.24 ▶ An abstract state-value pair $\langle \mathbb{S}_1, v_1 \rangle$ is **dominated** by another pair $\langle \mathbb{S}_2, v_2 \rangle$ iff $\mathbb{S}_1 \preceq_{\mathbb{L}} \mathbb{S}_2$ and $v_1 < v_2$.

A second way to simplify value partitions is to capture the same semantics (i.e. the same state-value mapping) but using a different set of descriptions that is smaller, more compact, simpler or anything other, measured by some (space) complexity measure. One of the simplest methods is to replace each description in a value partition by its reduced form, though for many formalisms highly effective, other methods exist. The only requirement is that the state-value mapping is left unaltered (though going beyond that forms an initial approach to approximations).

Simplifications can be performed at the end of iterations (such as is shown in Figure 6.3d), but preferably simplifications are used throughout many steps in the process of IDP, because less components yield less computation per iteration.

A Complete IDP Algorithm. Now we are ready to present IDP in Algorithm 16. In a

Algorithm 16 STEP V: **intensional dynamic programming** in a state description language \mathbb{L} . Lines 6–12 implement $B_{\mathbb{L}}^a$ for all actions a . Together with line 13 they implement the full operator $B_{\mathbb{L}}^*$. See Figure 6.4 for a graphical representation of the algorithm.

```

1: STEP V: Intensional Dynamic Programming
2:  $\mathbb{V}^0 = \mathbb{R} = \{ \langle \mathbb{V}_1^0, v_1^0 \rangle, \dots, \langle \mathbb{V}_n^0, v_n^0 \rangle \}$ 
3:  $k = 1$ 
4: repeat
5:    $\mathbb{Q}_{\sim}^{k+1} := \emptyset$ 
6:   for each  $\langle \mathbb{V}_i^k, v_i^k \rangle \in \mathbb{V}^k$  do
7:     for each  $\langle \mathbb{C}, a, \mathbb{E} \rangle \in A$  do
8:       for each  $\langle \mathbb{E}_j, p_j \rangle \in \mathbb{E}$  do
9:          $\mathbb{O} = \text{overlap}(\mathbb{V}_i^k, \mathbb{E}_j)$ 
10:         $\mathbb{Q}_{\sim}^{k+1} := \mathbb{Q}_{\sim}^{k+1} \cup \{ \langle \text{wp}_j(\mathbb{O}, a), a_j, (\gamma \cdot p_j \cdot v_i^k) \rangle \}$ 
11:    $\mathbb{Q}^{k+1} := \text{combine}(\mathbb{Q}_{\sim}^{k+1})$ 
12:    $\mathbb{V}^{k+1} := \mathbb{R} \boxplus \max_A \mathbb{Q}^{k+1}$ 
13:    $\mathbb{V}^{k+1} := \text{simplify}(\mathbb{V}^{k+1})$ 
14:    $\Delta := \arg \max_{\langle \cdot, \Delta \rangle} | \mathbb{V}^{k+1} \boxminus \mathbb{V}^k |$ 
15:    $k := k + 1$ 
16: until  $\Delta < \sigma$ 
    
```

similar way as we have done for set-based DP (see Equation 6.5) we can devise a closed form of a Bellman backup operator for the intensional setting.

DEFINITION 6.1.25 ▶ Let M be an MDP based on a state description language \mathbb{L} (with state space S and action space A). Furthermore let $\mathbb{V}^k = \{ \langle \mathbb{V}_1^k, v_1^k \rangle, \dots, \langle \mathbb{V}_n^k, v_n^k \rangle \}$ be an

intensional value function in \mathbb{L} over S .

The **intensional Bellman backup operator** \mathbf{B}_{\boxplus}^* in \mathbb{L} is defined as follows:

$$\begin{aligned} \mathbb{V}^{k+1} &= (\mathbf{B}_{\mathbb{L}}^* \mathbb{V}^k) \\ &= \text{simplify} \left(\mathbb{R} \boxplus \max_A \left(\underbrace{\text{combine} \bigcup_{k,a,\mathbb{E}_j} \left\{ \left\langle \text{wp}_j(\text{overlap}(\mathbb{E}_j, \mathbb{V}_i^k), \mathbf{a}), a_j, (\gamma \cdot p_j \cdot v_i^k) \right\rangle \right\}}_{\mathbf{B}_{\mathbb{L}}^a, \forall a \in A} \right) \right) \end{aligned} \quad (6.8)$$

For intensional value functions, $\mathbf{B}_{\mathbb{L}}^*$ thus replaces \mathbf{B}_{\boxplus}^* . Furthermore, $\mathbf{B}_{\mathbb{L}}^a$ is the intensional version of \mathbf{B}_{\boxplus}^a and is highlighted in the equation above. The backup operator $\mathbf{B}_{\boxplus}^\pi$ is replaced by $\mathbf{B}_{\mathbb{L}}^\pi$ that intersects the outcome of $\mathbf{B}_{\mathbb{L}}^a$ with an intensional policy.

Now, based on the correspondence between intensionally specified sets, and the actual sets modeled by them, and in addition all the operators we have described that model set-based operations, we can state the following. Let V^k be a set-based value function over a state space S and let \mathbb{V}^k be a value function in language \mathbb{L} such that $V^k(s) = \mathbb{V}^k(s)$ for all $s \in S$. Then,

PROPOSITION 6.1.4 ► for all states $s \in S$ the following holds:

$$V^{k+1}(s) = \left(\mathbf{B}_{\mathbb{L}}^* V^k \right)(s) = \left(\mathbf{B}_{\boxplus}^* \mathbb{V}^k \right)(s) \quad (6.9)$$

Proof. Through operations 1 to 5 we have upgraded all set-based operations in set-based DP to their intensional versions, thus rendering this a direct consequence of Proposition 6.1.2. The only difference with the set-based setting is that if the language \mathbb{L} is not expressive enough to generate partitions, and it is the case that for at least one $s \in S$ there are two value rules $\langle \mathbb{V}_i^k, v_i^k \rangle$ and $\langle \mathbb{V}_j^k, v_j^k \rangle$ ($i \neq j$) such that $s \in \llbracket \mathbb{V}_i^k \rrbracket$ and $s \in \llbracket \mathbb{V}_j^k \rrbracket$. Regressing both \mathbb{V}_i^k and \mathbb{V}_j^k can generate multiple rules with different values. Due to the max-covering semantics (see Definition 6.1.18), the value of a state s maintains the maximum value that can be achieved in the current horizon. \square

With this result it is easy to see that IDP converges to a value function that is optimal at the individual state level. The following proposition is a direct upgrade of Proposition 6.1.3.

PROPOSITION 6.1.5 ► The **optimal value function** $\mathbb{V}^* = \{ \langle \mathbb{V}_1^*, v_1^* \rangle, \dots, \langle \mathbb{V}_n^*, v_n^* \rangle \}$ satisfies the following **fixed point** definition:

$$\mathbb{V}^* = \left(\mathbf{B}_{\mathbb{L}}^* \mathbb{V}^* \right) \quad (6.10)$$

We will now review some current existing examples of IDP algorithms.

6.1.5.4 EXAMPLES OF IDP ALGORITHMS

The rather abstract nature of Algorithm 16 can be, and has been, instantiated in various forms. There are several classes of algorithms that we will discuss, which all mainly differ in the specifics of the state description language that is used. Several authors have

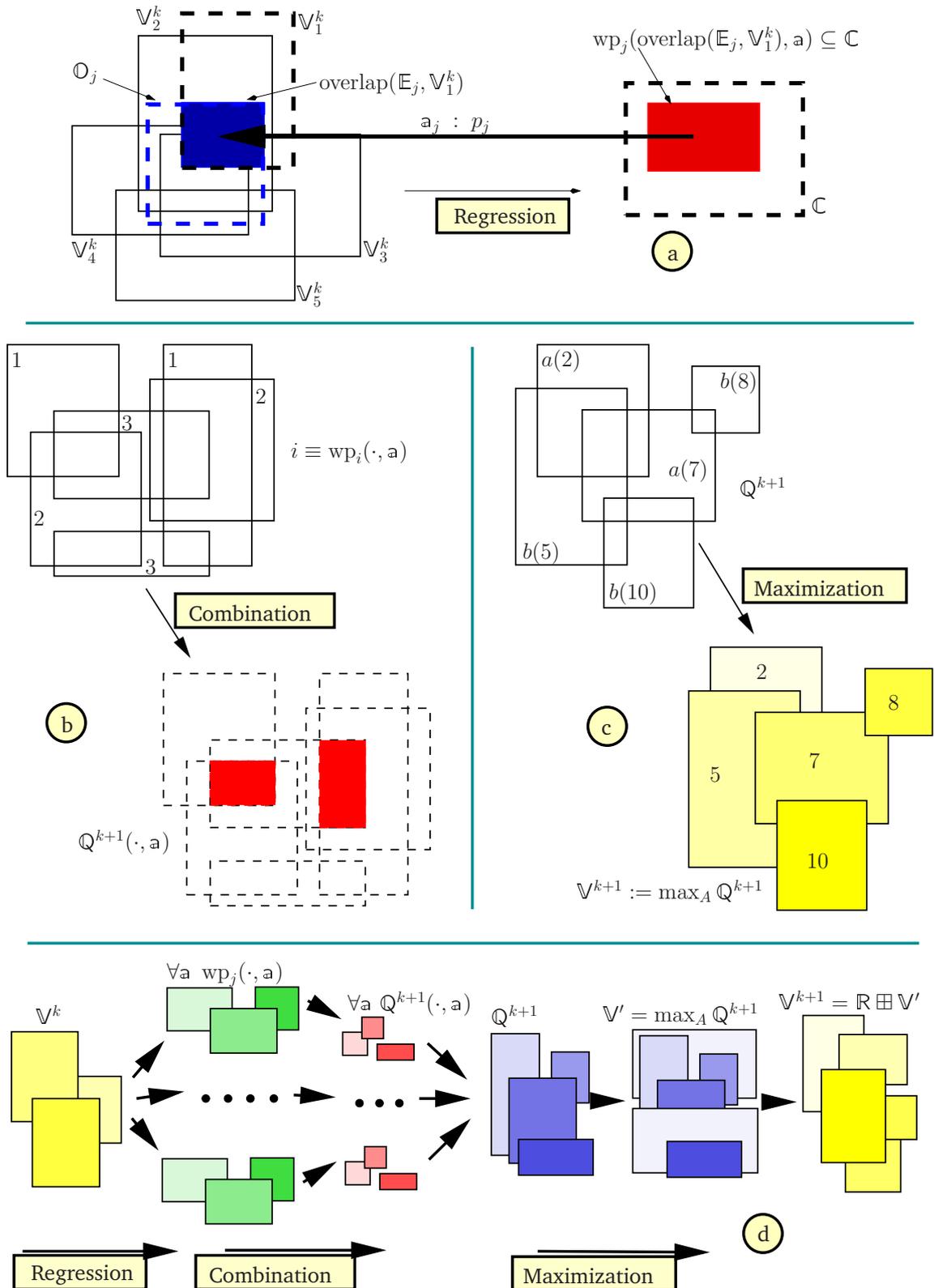


Figure 6.4: Intensional Dynamic Programming (see text for explanation).

realized that – in principle – their work can be generalized to generic languages, as is witnessed by the following quotes from two core texts that we have already encountered in Section 3.5.2.

Dietterich and Flann (1997, p. 182–183): *”It is important to note that although this algorithm is expressed in terms of rectangles, **the same approach is suitable to any problem where regions can be represented intensionally.** For example, expressions in propositional logic are equivalent to hyperrectangles in a high-dimensional space with one dimension for each proposition symbol. Expressions in first-order logic provide an even more powerful representation for regions. The computational geometry algorithms referred to above do not apply to these higher-dimensional representations, but the OFFLINE-RECT-DP algorithm works without modification as long as a data structure can be implemented that represents a collection of regions with attached priorities and that can efficiently determine the region of highest priority that contains a given point.”* (emphasis added)

Boutilier *et al.* (2000a, p. 103): *”We note that decision-theoretic regression is a general concept whose applicability is not restricted to decision-tree representations of transition functions, value functions and the like. **The same principles apply to any structured representation as long as one can develop a suitable regression operator for that representation.** To wit, the SPUDD system (Hoey *et al.*, 1999) applies the same decision-theoretic regression techniques to the solution of MDPs by value iteration, but does so using algebraic decision diagrams to represent inputs and output. Because these representations are often more compact than decision trees, the performance is considerably better than that of SPI; but it adopts the same general conceptualization of the problem...”* (emphasis added)

The first says that one only needs suitable data structures and a computable ordering on abstract states. The second says that one needs at least a regression operator. In general, based on the preceding sections, we can distinguish a number of important aspects of formal systems for IDP.

First of all, there is the aspect of *semantics*, i.e. the structure of states in the environment. The semantics of the system concerns the type of *interpretations* that form the states. These interpretations can range from symbolic units, to propositional or first-order structures to even more complex ones involving time or higher-order structures (see also Chapters 2 to 4).

Second, the *syntax* of the formal system determines the expressivity and reasoning patterns. How, and how precise, state sets can be described, is determined by the syntax of the state description language. Of particular interest is whether *partitions* can be described declaratively, which, for example, is important for the description of value functions. The complexity of reasoning (e.g. deductive procedures) is important for practical use in IDP algorithms. Deciding coverage or computing with language expressions amounts to syntactic reasoning, and its complexity is determined by the language used. Often, *language fragments* can be used that provide a better trade-off between a rich language and fast evaluation (see also Chapter 4). The IDP algorithm involves operations such as intersection, simplification and other language operations that – semantically – perform set-based computations. Often these can be computed more efficiently by using efficient representations, (e.g. ADDs instead of trees, see Hoey *et al.*, 1999). An ordering on language

expressions, and efficient ways to compute it, can improve on IDP algorithms, for example in simplification routines.

Third, the way *actions* are represented, how the frame problem is solved (or even a richer domain theory is formalized), and also how efficient their (operational) semantics is, are very important for computationally expensive IDP algorithms. Most importantly, the way in which *regression* is defined and computed, is important for IDP. For example, in situation calculus (see Section 4.4.2.2) it is a common operation, though in this chapter we will see that for STRIPS-like languages, additional means are necessary.

These characteristics can be used to compare and evaluate IDP algorithms. In the following, we will briefly give a number of examples from the literature. All of them implement IDP in propositional (either discrete or continuous) domains, though some may deviate slightly from Algorithm 16. Some have carried the idea of IDP further, by taking it to multi-agent decision making. These systems intersect value partitions for multiple agents, and use a min-max criterion to compute values for all agents in a way similar to game theory (Yee *et al.*, 1990; Dietterich and Flann, 1997; Finzi and Lukasiewicz, 2004a).

IDP EXAMPLE I: Explanation-Based Reinforcement Learning. A first development in IDP is *explanation-based* RL (EBRL) (Dietterich and Flann, 1995, 1997) and we have described it briefly in Section 3.5.2. The inspiration for this work, and its hierarchical extension (Tadepalli and Dietterich, 1997), is the similarity between Bellman backups and *explanation-based generalization* (EBG) in *explanation-based learning* (EBL) (e.g. see Ellman, 1989; Minton *et al.*, 1989, also for the source of the term *justified generalization*).

The language used in EBRL consists of *rectangular regions* in a d -dimensional space. The most common application is navigation in (discrete) $2D$ grid worlds. Rectangles are aligned with the squares in the grid, such that each one can be represented by the two states in two opposite corners. Such rectangles allow for a spatial interpretation (i.e. states covered by a rectangle are geometrically close) and this induces a natural ordering. Weakest preconditions can be computed quite easily from the action definitions. Value functions are represented as a set of (overlapping) rectangle-value pairs, resembling the max-coverings in Definition 6.1.18, with a default ∞ -valued rectangle³¹ covering the complete state space. Making use of algorithms from computational geometry, finding the top rectangle at a particular point using the ordering has a complexity $O(\log^3 n)$ (n is the number of rectangles). If ρ denotes the average number of states covered by a rectangle, then the complexity of inserting a new rectangle is $O(\log^3(\frac{n}{\rho}) + \rho \log^2(\frac{n}{\rho}))$. Thus, these and other operations (e.g. computing overlaps of rectangles) are relatively inexpensive, though the expressivity of the language is limited in other³² domains than grid worlds. If states that have to be aggregated, are separated by great distances, many rectangles are needed to store the value function.

Dietterich and Flann (1997) introduce several algorithms, varying between *deterministic* and *stochastic* domains, between *point-based* (i.e. state-based) and *region-based*, and between *online* and *offline*. The most complex algorithm (STOCHASTIC-OFFLINE-RECT-DP, doing region-based backups in stochastic domains without trajectory sampling) can be

³¹EBRL uses a cost-based reward framework, trying to *minimize* the total reward (i.e. cost) intake, which explains the infinite value instead of e.g. a value 0.

³²Dietterich and Flann (1997) also performed experiments in CHESS endgames, where rectangular regions correspond to particular areas on the playing board. There each region contains a two-dimensional rectangle for each playing piece on the board.

seen as an instantiation of Algorithm 16. One of the differences is that the STOCHASTIC-OFFLINE-RECT-DP-algorithm performs some operations in a different order (e.g. the algorithm first maximizes, and then performs regression to compute a new Q -value function, in one loop). Another difference is that it uses a separate *priority queue* and a value function to store and to prioritize which regions are used for updates, rendering the algorithm *asynchronous* though still complete sweeps are computed. The altered order is beneficial in combination³³ with the specific storage and retrieval of rectangles in this algorithm, decreasing the amount of shattering.

EBRL is based on earlier³⁴ work on T2 by Yee *et al.* (1990) who use EBG and goal regression in the context of TD-learning. (Yee *et al.*, 1990, p. 1): *"In contrast, T2 takes advantage of its knowledge of the problem domain to generalize the descriptions of states before they are cached."* Unlike EBRL and IDP in general, T2 does not exactly aggregate operations in existing MDP solution techniques (e.g. as in IDP), but instead it employs heuristics to generalize actual experience to store generalized concept-value pairs. Still, it marks some of the first efforts to perform generalized value backups.

IDP EXAMPLE II: Factored Representations and Structured Algorithms. A second example are the structured solution algorithms SPI and SVI by Boutilier *et al.* (2000a), which we have discussed in some detail in Section 3.5.2. Whereas EBRL is targeted at goal-based reward functions, SPI and SVI work with general reward functions. The state description language consists of factored, propositional representations in the form of decision trees (other representations such as PSTRIPS have been used before, e.g. see Boutilier and Dearden, 1994; Dearden and Boutilier, 1997). Action dynamics are represented using Bayesian networks, and reward and value functions, as well as policies, are represented as trees. Each such tree forms a *partition* of the state space, and various operations on trees (such as intersection, addition and simplification) are well-defined. Decision-theoretic regression can be formalized over trees, such that the regression of a tree-based representation of V^k through an action a is a tree-based representation of $Q^{k+1}(\cdot, a)$. Regression traverses V^k and computes the complete structure of the weakest precondition recursively for each action at once, due to the fact that the dynamics of each action is contained in one structure. For each variable f in the state representation, regression is used to find those circumstances in which the action will make f true. Regression through a policy (enabling to perform tree-based policy iteration) is defined along the same lines as in IDP.

Boutilier *et al.* (2000a) also discuss issues such as approximation and synchronic effects, both in the context of tree representations. An interesting asynchronous DP extension of the approach is *structured prioritized sweeping* based on the work by Andre *et al.* (1998) and the following idea (Dearden, 2001, p. 84): *"In SPI, the Regress operator is used to perform the step of value iteration over the whole value tree. However, there is no reason why the operator can't be applied on much smaller parts of the state space."* This enables model-free methods to compute value functions while still using structured backups, i.e. without complete state space enumeration. Though not described by the authors, there

³³In fact, Dietterich and Flann (1997, p. 189–190) propose another algorithm in which the actions are converted into deterministic ones by only taking the highest-valued outcome. Then, synchronous DP is done with the stochastic actions. This approach creates larger rectangles, which is better.

³⁴Yee *et al.* (1990) write that they, in turn, were influenced by the work by Samuel (1959), in which states were memorized completely. In 1990 the work by Sutton (1988) was very recent, and not much work on abstraction in RL had been done.

are many connections with the EBRL framework of the previous paragraph, but in general, all fit in the framework of IDP.

Very much related to these algorithms is the *model-minimization* (MM) framework by Givan *et al.* (2003) which we have described in Section 3.5.2. Remember that the end product of IDP is a set of regions that aggregate states that are 'the same'. These clusters form a structure that is a reduced version of the original MDP. MM consists of several methods to find such reduced MDPs by adapting techniques for automata minimization, focusing completely on the structural part of the model (i.e. the aggregates). Usually the algorithm starts with one aggregate containing the complete state space, and iteratively the state space partitioning is *refined* by *splitting* aggregates. One splitting method that comes very close to IDP is the R-SPLIT, which refines the current value partition by regression. Givan *et al.* (2003, p. 192–197) discuss at length the correspondence between MM using regression and algorithms such as SPI and SVI. The main difference is that the latter perform parameter and structure computations simultaneously, whereas MM is only concerned with the structural part. One difference that applies specifically to SPI is that it performs aggregation relative to different specific policies encountered whereas MM relative to all policies (states must be separated if they differ under *any* policy).

IDP EXAMPLE III: Continuous MDPs and POMDPs. A conceptually simple extension of the rectangular regions used by EBRL are the (hyper)-rectangular partitions of continuous state spaces by³⁵ Feng *et al.* (2004), that can also be seen as a multi-dimensional extension of the work on time-dependent MDPs by Boyan and Littman (2000). In a continuous MDP over a state space X , we are interested in solving the following Bellman equation:

$$V^{k+1}(\vec{x}) = \max_{a \in A} \left[R(\vec{x}) + \int_X T(\vec{x}' | \vec{x}a) V^k(\vec{x}') d\vec{x}' \right] \quad (6.11)$$

Actions shift a complete region \square by some distance δ :

$$\forall x, y \in \square : T_a(x + \delta x, x) = T_a(y + \delta x, y)$$

Value functions and policies can be represented as a partition consisting of hyperplanes in some d -dimensional space:

DEFINITION 6.1.26 ► A **rectangular partition** of the state space $[0, 1)^d$ is a finite set of rectangles $\boxplus = \{\square_1, \square_2, \dots, \square_k\}$, where $\square_i = \Pi[x_i^{\text{low}}, x_i^{\text{high}})$, such that $\bigcup_{1 \leq i \leq k} \square_i = [0, 1)^d$, and $\square_i \cap \square_j = \emptyset$, iff $i \neq j$. A function $f : [0, 1)^d \rightarrow \mathbb{R}$ is **rectangular piecewise constant** (RPWC), if there exists a rectangular partition $\boxplus = \{\square_1, \square_2, \dots, \square_k\}$ such that for $\forall i, 1 \leq i \leq k$ and $\forall x, y \in \square_i, f(x) = f(y)$.

RPWC functions are stored using kd -trees (splitting hyperplanes along k axes) for efficient manipulation. Note that in IDP this is equivalent to a set of abstract states that form a partition consisting of infinite state sets. Computing intersections and merging rectangles is done relatively to the kd -tree such that the structure is maintained throughout the whole process, i.e. if V^n is RPWC then V^{n+1} is also RPWC.

³⁵An alternative description of the same approach was given in the lecture notes of Richard Dearden's IJCAI tutorial in 2005.

Feng *et al.* (2004) extend their framework to piece-wise *linear* and *convex* (PWLC) value functions. This enables more complex reward functions, where values can vary (linearly) throughout regions. Feng *et al.*'s approach can be further extended by allowing for more complex transition function definitions, such as continuous transition probability distributions (Li and Littman, 2005) and phase-type distributions (Marecki *et al.*, 2006). Li and Littman's approach shows an interesting method that first efficiently computes a PWLC value function representation using a structured Bellman backup, and then approximates it using a PWC value function before continuing with the next VI iteration.

The PWLC structure is also used in POMDP solution techniques where value functions over the belief space are expressed by a set of vectors $\Gamma = \{\alpha_0, \alpha_1, \dots, \alpha_m\}$ and the belief state value is $V(b) = \max_{\alpha \in \Gamma} \sum_{s \in S} \alpha(s)b(s)$, where $b(s)$ is the probability the belief state assigns to state s (see Kaelbling *et al.*, 1998; Aberdeen, 2003; Spaan, 2006). Furthermore, operations on these α -vectors (including pruning of dominated vectors) are well-defined in the POMDP literature. It is beyond the scope of this book to further describe the intimate relationships between factored MDP representations, structured algorithms and (exact) algorithms for POMDPs. But, there are many connections, both on the conceptual level (e.g. the regression step in IDP must be extended with an additional step from observations to states) and the technical level. As an example of this, quoting Boutilier *et al.* (2000a, p. 104): "*Investigations into the application of SPI to POMDPs is reported in (Boutilier and Poole, 1996), where vectors corresponding to the usual piecewise linear representations of value functions for POMDPs are treated as decision-trees, produced by decision-theoretic regression.*" Many other examples of structured POMDP algorithms have been studied (e.g. Boutilier and Poole, 1996; Poole, 1997a; Geffner and Bonet, 1998; Wang and Schmolze, 2005), and recently initial progress was made in the solution of *relational* POMDPs (Wang, 2007) that upgrades the classical *incremental pruning* algorithm for POMDPs to the first-order case (see at the end of this chapter).

6.1.5.5 DISCUSSION

Whereas set-based DP is merely a theoretical tool that highlights structure in DP algorithms, IDP is a more general format that can be used – once instantiated with a suitable formal language – to solve large sequential decision making problems. In other words, set-based DP provides a *semantics* for various IDP systems that differ in the *syntax* of their state description languages. IDP algorithms function on both the structural and parameter level, i.e. in the terminology of Chapter 3, IDP algorithms belong to PIAGET-3.

Especially when standard logical languages are used, IDP algorithms can be interpreted as a kind of *decision-theoretic* logic, where state values are *deduced* from the action definitions and the reward function. For action logics this is entirely true, yet for action formalisms this is only true when we adopt a slightly less strict definition of deduction (see Section 4.4 on this). The Bellman backup operator \mathbf{B}_{\perp}^* in Equation 6.8 functions as a deduction rule and each value $V^*(s)$ is uniquely defined by the fixed point of $(\mathbf{B}_{\perp}^* R)^*$. For systems based on continuous spaces and hyperrectangles, it gets harder to view them as decision-theoretic logics, though we have shown that they are based on exactly the same reasoning patterns.

Now that we have formalized set-based DP as giving semantics to IDP, and we have mentioned a number of IDP systems, we can list a number of *requirements for any state description language to implement an IDP algorithm*:

- **Formal Syntax and Semantics:** A state description language must be specified along with a mapping to the semantics that consists of subsets of a state space. The syntax corresponds to the nature of the states, which can vary from discrete symbols to continuous belief states or first-order structures, and simultaneously forms a language for expressing bias about the policy and value functions.
- **Operations:** Operations on abstract state descriptions that are (at least) needed in Algorithm 16, are *intersection* and *difference*. In some logical languages these correspond to logical connectives such as AND (\wedge), but e.g. for rectangular regions in n -dimensional spaces they must be computed using specialized algorithms.
- **Actions and Regression:** Based on the formal syntax and semantics, action definitions must specify which actions exist and how they transform abstract state descriptions (assuming the Markov property). Actions can be axiomatized or governed by rules and additional, operational means, and furthermore interaction with a background theory can be formalized. A regression operator must be provided that may use the action definitions to compute descriptions of weakest preconditions.
- **Simplification and Efficient Data Structures:** The above three components are enough to create an IDP system, though it will not scale up very far. In order to keep the amount of descriptive components manageable, one should make use of (or define) reduction or simplification mechanisms for a given language, or use efficient data structures (e.g. ADDs, decision lists, trees, etc.).

A wide variety of variations on the scheme of IDP is possible. For example, we have seen representation-dependent modifications to the order of the computation, online algorithms utilizing structured backups and in the remainder of the chapter we will see combinations with search techniques. Most of the techniques outlined in Chapter 2 and especially Section 2.5.2 – such as asynchronous DP algorithms, and the use of heuristics, envelopes and search – can be adapted to the IDP framework and are open research issues. In fact, anywhere structured, full (or even partial) Bellman backups are used, Equation 6.8 can be employed.

One of the most prominent areas for improvement and extension is *approximation*. Depending on the expressivity of the language, at some point it becomes necessary to give up on exact representation of the value function and approximate it. The main source of this problem is known as the *utility problem* in EBL (see Ellman, 1989; Minton *et al.*, 1989). When the value function is represented using too many abstract states, one problem is that one iteration of the IDP algorithm requires much computational effort. But also, more abstract states can make the system spend more time matching all these states (and computing many nearly irrelevant low-utility abstract states) than searching for a solution in the original state space. Simple approximations involve aggregating states within bounds (as opposed to exactly) but many other types of approximations will involve model-free algorithms and *inductive* learning, especially when the operator model is incorrect or incomplete.

A second direction for extensions of IDP are POMDPs, especially for relational domains. We have described how continuous MDPs can be solved through IDP, and POMDPs induce such continuous MDPs. Still, the most promising approaches take existing structured algorithms for MDPs (such as IDP) and extend these beyond the Markov assump-

tion to approach POMDPs. Instead of exact approaches to POMDPs along these lines, *point-based* approximations are very useful (and often anytime approximations). Relational point-based extensions are still abundant, though one of the first algorithms for relational POMDPs was recently proposed by Wang (2007). In the remainder of this chapter we will describe in detail one relational instantiation of IDP, REBEL, and survey all related methods.

6.2. A Relational State Description Language

In this section we develop a state description language for RMDPs, denoted *Markov decision programs*. We give a complete description of the expressivity and properties of the language, and we describe all operations necessary to employ the generic IDP structure in relational domains in Section 6.3. We restrict ourselves to a subset of FOL. For the specification of the transition function, our language coincides with a standard first-order, probabilistic STRIPS language. State descriptions are limited to positive atoms only, augmented with syntactic constraints on variable bindings. At least two notable extensions to state description languages will be introduced in this section. One is that we work with an *open world semantics*, meaning intuitively that a given Markov decision program may apply to a whole range of (different-size) problems. The other is that our notion of a *state description language* is taken slightly more general because *actions* play a more important role in the descriptions than in the propositional case, due to variable dependencies between states and the action's parameters.

6.2.1 Abstract States

An abstract state models a set of interpretations of the underlying RMDP. An abstract state defines which relations should hold in each of the states it covers. The use of variables admits abstracting over specific domain objects as well. We start with a first-order logical alphabet containing predicates, constants and variables (see further Section 4.2.1).

DEFINITION 6.2.1 ► An **abstract state** is a conjunction $Z \equiv Z_1 \wedge \dots \wedge Z_m$ of logical atoms, i.e., a logical query. (We will usually use the equivalent notation $(Z_1 \dots Z_m)$.)

In addition to the atoms describing an abstract state, we also employ *constraints* on variables. These can be expressed by an additional binary relation \neq . For ease of readability, we will use an infix notation. More details on the constraints can be found in Section 6.2.4.

A conjunction is implicitly *existentially quantified*, i.e. an abstract state $Z \equiv \text{on}(X, Y)$ should be seen as $\exists X \exists Y Z$ which reads as *there are two blocks, denoted X and Y and block X is on block Y*. Furthermore, an abstract state Z covers a concrete (ground) state z , i.e. an interpretation, iff $z \models Z$. We use Herbrand-style semantics, where ground states (i.e. Herbrand interpretations) are in the language (see Chapter 4 on these matters). In addition, we employ an *open world assumption* (OWA), which means that abstract states are to be considered *partial descriptions* of ground states.

EXAMPLE 6.2.1 ► Consider e.g. the state $z \equiv \text{cl}(a), \text{cl}(b), \text{on}(a, c)$ in BLOCKS WORLD. An abstract state Z is, e.g., $\text{cl}(X)$. Now $z \models Z$, i.e. Z covers the state z . In fact, Z covers all states that are interpretations in which there exists *something* that is clear. The OWA makes it possible to decide on coverage without knowing how many objects we have beforehand;

as long as there is *some* clear block in a state, it is covered by \mathbb{Z} . Thus, depending on the number of blocks available in some concrete domain instantiation, \mathbb{Z} may have few or many models, and in fact even an infinite amount of them.

Abstract states are not very expressive. For example, they are functor-free and they are not capable of directly expressing *negation* or *disjunction*. Another important aspect is that it is not possible to express *universal quantification*. However, in the following we will use sets of abstract states to express some form of disjunction, and limited versions of negation will be used for constraints on variable bindings. In the experimental section we will use a specific version of negation for designated binary atoms and later we will elaborate on universally quantified goals.

Ordering and Coverage. Abstract states are (partially) ordered by θ -*subsumption*, which we have defined in Section 4.3.2.2. An abstract state \mathbb{Z}_1 θ -subsumes abstract state \mathbb{Z}_2 (denoted $\mathbb{Z}_2 \preceq_{\theta} \mathbb{Z}_1$) iff there exists a substitution θ such that all literals in $\mathbb{Z}_1\theta$ occur in \mathbb{Z}_2 . Deciding subsumption (between states) will be the main reasoning operation, a computationally feasible substitute for a generic *proof* relation \vdash . A ground state z is *covered* by abstract state \mathbb{Z} , iff $z \preceq_{\theta} \mathbb{Z}$. Although subsumption is decidable, it is computationally expensive. According to Kietz and Lübke (1994), the (relative) inefficiency of \preceq_{θ} is caused by the nondeterminism of choosing θ . Some efficient versions of θ -subsumption have been developed that for example operate on the variable chains in abstract states (Kietz and Lübke, 1994; Maloberti and Sebag, 2004; Skvortsova, 2006b) but in the current chapter we use a standard PROLOG membership test.

Operations. Because our language is a small subset of FOL, many standard operations exist (see Chapter 4). Of particular importance for the algorithms in this chapter are the *greatest lower bound* (glb), that can be used to compute *overlaps*, and *reduction*, for simplification of abstract states. Furthermore, we make use of the *most general unifier* (mgu) (see Definition 4.2.10).

DEFINITION 6.2.2 ► The **greatest lower bound** (glb) of two states \mathbb{Z}_1 and \mathbb{Z}_2 is the most general conjunction \mathbb{Z}' that is subsumed by both \mathbb{Z}_1 and \mathbb{Z}_2 . Another way of seeing the glb is as the **overlap** between two model sets, i.e. $\llbracket \mathbb{Z}_1 \rrbracket \cap \llbracket \mathbb{Z}_2 \rrbracket$.

Simplification of abstract states can be defined relative to the θ -subsumption ordering. As we have explained in Section 4.3.2, the subsumption lattice is defined over equivalence classes of descriptions that are equal under θ -subsumption, and a reduced clause is the smallest description of its class. Non-reduced descriptions contain *redundant* atoms that can be discarded without changing the coverage of the description. For example, the abstract state $(\text{clear}(X), \text{clear}(Y))$ can be reduced to $(\text{clear}(X))$ because they subsume each other (i.e. X and Y can be chosen to be equal). Redundant atoms are often constructed by operations on abstract states (most notoriously is the *least general generalization* construction, see later in this chapter), and it is important to keep descriptions small as most operations have a complexity that is proportional to the number of atoms.

Reductions are not often described in the literature, though many (practical) systems are dependent on procedures to keep descriptions manageable. Some explicit mentioning of how to compute reductions are described by van Laer and De Raedt (2001a); van Laer (2002) and Nienhuys-Cheng and de Wolf (1997), and in this chapter we use the FASTCON-

DENSE algorithm by Gottlob and Fermüller (1993). The general idea – described in Algorithm 17 – is to consider each individual atom and to use subsumption to check whether it is redundant (and can be removed). Let \mathbb{C} be an abstract state, then the complexity of computing the reduced form of \mathbb{C} is sub-linear in $|\mathbb{C}|$, because it makes at most $|\mathbb{C}|$ calls to a subsumption checker, which is computationally hard. This also shows that simplification could be made faster by implementing a faster subsumption checking algorithm. Abstract states are fairly simple constructs. Due to that, and due to correspondence with

Algorithm 17 The algorithm FASTCONDENSE by Gottlob and Fermüller (1993), slightly adapted for conjunctive states. Matching is done by checking whether a substitution exists.

1: [REDUCE] : *Simplification of Abstract States*

Require: the input is an abstract state \mathbb{C}

Require: the output is a reduction of \mathbb{C}

```

2:  $\mathbb{D} := \mathbb{C}$ 
3:  $\mathbb{E} := \mathbb{C}$ 
4: while  $E \neq \emptyset$  do
5:   choose some  $\mathbb{L} \in \mathbb{E}$ 
6:   if  $\exists \mathbb{L}' \in \mathbb{D} - \{\mathbb{L}\}$  such that  $\mathbb{L}$  matches  $\mathbb{L}'$  then
7:     if  $\text{subsumes}(\mathbb{C}, \mathbb{D} - \{\mathbb{L}\})$  then
8:        $\mathbb{D} := \mathbb{D} - \{\mathbb{L}\}$ 
9:        $\mathbb{E} := \mathbb{E} - \{\mathbb{L}\}$ 
10: return  $\mathbb{D}$ 
    
```

representations frequently used in ILP (see Section 4.3.2), most necessary operations are naturally defined, and computationally feasible. Furthermore, they are easily incorporated into STRIPS action definitions in the next section.

6.2.2 Abstract Actions

An action defines a mapping from one abstract state to another, i.e. from a set of states to another set of states. An abstract action defines a *probabilistic* action operator by means of a set of deterministic action operators, under influence of a probability distribution over the possible outcomes of the action.

DEFINITION 6.2.3 ► An **abstract action** is a tuple $\langle \mathbb{B}, \mathbf{a}, [\langle \mathbb{H}_1, p_1 \rangle, \dots, \langle \mathbb{H}_m, p_m \rangle] \rangle$ where \mathbf{a} is an atom representing the name and the arguments of the action and \mathbb{B} is an abstract state denoting the preconditions of \mathbf{a} . \mathbb{H}_i is the i -th possible outcome of \mathbf{a} with probability $0 \geq p_i \geq 1$. The precondition of the stochastic action \mathbf{a} is $\text{pre}(\mathbf{a}) \equiv \mathbb{B}$ and the postcondition of the i -th rule of \mathbf{a} , is $\text{post}_i(\mathbf{a}) \equiv \mathbb{H}_i$. It holds that $\sum_{i=1}^m p_i = 1$. Furthermore, $\text{vars}(\mathbf{a}) = \text{vars}(\text{pre}(\mathbf{a})) \cup_i \text{vars}(\text{post}_i(\mathbf{a}))$.

The procedural semantics of the action definition are:

If the current state \mathbf{z} is subsumed by $\text{pre}(\mathbf{a})$, i.e., $\mathbf{z} \preceq_{\theta} \text{pre}(\mathbf{a})$, then taking action \mathbf{a} will result in $[\mathbf{z} \setminus \text{pre}(\mathbf{a})\theta] \cup \text{post}_i(\mathbf{a})\theta$ with probability p_i .

We assume that if the preconditions are fulfilled, all outcomes are possible (but the actual outcome is "under Nature's control" (see Section 4.4). In addition, we assume that all outcomes $\text{post}_i(\mathbf{a})$ for an abstract action \mathbf{a} are disjoint, given $\text{pre}(\mathbf{a})$.

The i -th rule of an abstract action is denoted $\langle \text{pre}(\mathbf{a}), \mathbf{a}, p_i, \text{post}(\mathbf{a}) \rangle$. We will mainly use the alternative *rule* (or, *clausal*) representation:

$$\text{pre}(\mathbf{a}) \xrightarrow{p_i : \mathbf{a}} \text{post}_i(\mathbf{a})$$

An action models a set of transitions of the underlying (ground) RMDP. More precisely, it models a transition between *sets* of states.

EXAMPLE 6.2.2 ▶ As an illustration, consider

$$\begin{array}{ccc} \text{cl}(X), \text{cl}(Y), \text{on}(X, Z), & \xrightarrow{0.9 : \text{move}(X, Y, Z)} & \text{on}(X, Y), \text{cl}(X), \text{cl}(Z), \\ X \neq Y, Y \neq Z, X \neq Z & & X \neq Y, Y \neq Z, X \neq Z \end{array}$$

$$\begin{array}{ccc} \text{cl}(X), \text{cl}(Y), \text{on}(X, Z), & \xrightarrow{0.1 : \text{move}(X, Y, Z)} & \text{cl}(X), \text{cl}(Y), \text{on}(X, Z), \\ X \neq Y, Y \neq Z, X \neq Z. & & X \neq Y, Y \neq Z, X \neq Z. \end{array}$$

which moves block X on Y with probability 0.9. With probability 0.1 the action fails, i.e., we do not change the state. Applied to $z \equiv \text{cl}(a), \text{cl}(b), \text{on}(a, c)$ the action tells us that $\text{move}(a, b, c)$ will result in $z' \equiv \text{on}(a, b), \text{cl}(a), \text{cl}(c)$ with probability 0.9 and with probability 0.1 we stay in z .

This type of action definition implements a kind of first-order version of a probabilistic STRIPS operator (Hanks and McDermott, 1994). A superficial difference is that our operators do not have separate *preconditions* and *add* and *delete* lists. However, for an abstract action \mathbf{a} one can view $\text{pre}(\mathbf{a})$ as a *precondition* which is also a *delete* list, whereas $\text{post}_i(\mathbf{a})$ can be seen as the *add* list. Based on Definition 6.2.3 we can define a regression procedure in Section 6.3.1.

6.2.3 Rewards

Reward functions are represented by *decision lists* (Rivest, 1987) of abstract states, augmented with a value, which means that rewards are associated with states. Extending the framework to work with action reward functions is straightforward, though additional reasoning steps are needed because states and actions are connected by variables.

DEFINITION 6.2.4 ▶ A **state reward function** \mathbb{R} is a finite list of rules of the form $\langle \mathcal{S}, r \rangle$ where \mathcal{S} is an abstract state and $r \in \mathbb{R}$. The **reward value** of abstract state Z is defined as

$$\mathbb{R}(Z) =_{\text{def}} \max\{r_i \mid \langle \mathcal{S}_i, r_i \rangle \in \mathbb{R} \wedge Z \preceq_{\theta} \mathcal{S}_i\}$$

State-based reward functions over conjunctive states are powerful enough to capture many interesting (goal-based) domains.

EXAMPLE 6.2.3 ▶ Consider the following two state reward functions \mathbb{R}_1 and \mathbb{R}_2 :

$$\begin{array}{ccc} \left\langle \begin{array}{l} \text{clear}(a), \text{clear}(b) \\ \text{clear}(a) \\ \text{true} \end{array}, \begin{array}{l} 10.0 \\ 5.0 \\ 0.0 \end{array} \right\rangle & & \left\langle \begin{array}{l} \text{clear}(a) \\ \text{clear}(b) \\ \text{true} \end{array}, \begin{array}{l} 9.0 \\ 6.0 \\ 0.0 \end{array} \right\rangle \end{array}$$

Both assign value 0.0 as a default value to all states. \mathbb{R}_1 assigns a value 10.0 to all states in which *both* blocks a and b are clear, but assigns a value 5.0 to all states in which *at*

least block a is clear. Note the subtle differences with \mathbb{R}_2 , because \mathbb{R}_2 assigns its highest value (9.0) to all states in which block a is clear, and assigns a value 6.0 to all states in which block a is not clear, but at least block b is. The difference between the two reward functions is due to the semantics of a `max-covering` using a set of rules.

State rewards are specified over queries, i.e., existentially quantified goals. Although these are simple, they are expressive enough to specify many interesting problems studied in (relational) RL and decision-theoretic planning communities such as *shortest-path problems* where the goal is to reach certain (abstract) states. When a goal state is entered, the process ends. In RL, episodic tasks are usually encoded using *absorbing states* (see also Section 2.2). We encode it by an *artificial* deterministic action `absorb`, defined as

$$\mathbb{G} \xrightarrow{1.0 : \text{absorb}} \mathbb{G}$$

where \mathbb{G} is an abstract state expressing a goal region, i.e. an absorbing abstract state. Assume e.g. $\mathbb{G} \equiv \text{on}(a, b)$, then all states that are subsumed by $\text{on}(a, b)$ transition only to themselves and generate only zero rewards. For example,

$$z \equiv \text{cl}(a), \text{cl}(b), \text{on}(a, c)$$

is not absorbing, but

$$z' \equiv \text{on}(a, b), \text{cl}(a), \text{cl}(c)$$

is.

6.2.4 Domain Theory and Constraint Handling

Up to now, we have assumed that the domain description comes with a language containing a number of predicates that are used in state descriptions, value functions and action definitions. However, most common domains will impose *constraints* on the language, based on certain *laws* of logic and nature. For example, even though abstract state descriptions such as $\text{on}(X, X)$ and $(\text{on}(X, Y), \text{clear}(Y))$ can simply be constructed in the BLOCKS WORLD, they do not make sense. In the first description a block would be on top of itself, whereas in the second, block Y is clear while at the same time block X is on top of it. In other words, the syntax used for describing states is often too general and covers ground states that do not belong to the environment, that is the underlying RMDP. Thus, restrictions must be placed on the syntactic level and for this we define so-called *domain rules*.

DEFINITION 6.2.5 ▶ A **domain rule** \mathbb{C} is a rule $\mathbb{H} \Rightarrow \mathbb{B}$ where either \mathbb{H} and \mathbb{B} are both conjunctions and the rule is called a **ramification**, or \mathbb{H} is a conjunction and $\mathbb{B} \equiv \text{false}$ and the rule is called an **integrity constraint**. The **application** of a rule $\mathbb{C} \equiv \mathbb{H} \Rightarrow \mathbb{B}$ to an abstract state Z , denoted $\mathbb{C}(Z)$, is the state Z' where

$$Z' := \begin{cases} Z \wedge \mathbb{B}\theta & , \text{if } \exists \mathbb{H}' \subseteq Z \text{ and } \mathbb{H}' \preceq_{\theta} \mathbb{H} \text{ and } \mathbb{B}\theta \not\subseteq Z \\ Z & , \text{otherwise} \end{cases}$$

A rule $\mathbb{H} \Rightarrow \mathbb{B}$ is **applicable** in a state Z if $Z \preceq_{\theta} \mathbb{H}$.

Domain rules have two different purposes. Rules of type $\mathbb{B} \Rightarrow \text{false}$ are only used to compute whether a description is legal according to the domain theory. When the pattern \mathbb{B} is found in a state description, the state is illegal. On the other hand, general rules of type $\mathbb{B} \Rightarrow \mathbb{H}$ are meant to *extend* a state description with new literals, similar to *ramifications* in general action theories (see Section 4.4.1.2).

EXAMPLE 6.2.4 ► Some example domain rules are the following.

$$\begin{array}{ll} \begin{array}{l} (\mathbb{L}_1) \\ (\mathbb{L}_3) \end{array} & \begin{array}{l} X \neq Y \Rightarrow Y \neq X \\ \text{on}(X, Y), \text{cl}(Y) \Rightarrow \text{false} \end{array} & \begin{array}{l} (\mathbb{L}_2) \\ (\mathbb{L}_4) \end{array} & \begin{array}{l} X \neq X \Rightarrow \text{false} \\ \text{on}(X, Y) \Rightarrow X \neq Y \end{array} \end{array}$$

The first rule \mathbb{L}_1 expresses the symmetry of 'not equal to'. The second \mathbb{L}_2 defines that it is not possible that any object is not equal to itself. \mathbb{L}_3 rules out the possibility that there is a block that is both clear and has a block on top of it. The last rule \mathbb{L}_4 states that when two blocks are on top of each other, they have to be two different blocks. Both \mathbb{L}_2 and \mathbb{L}_3 represent integrity constraints and add a `false` literal and render states illegal. The other two rules \mathbb{L}_1 and \mathbb{L}_4 represent ramifications, or *derived* literals, and *extend* an abstract state description with additional information. Together, rules form a so-called *domain theory*.

DEFINITION 6.2.6 ► A **domain theory** \mathbb{C} is a set of domain rules $\{\mathbb{C}_1, \dots, \mathbb{C}_m\}$. Let \mathbb{C} be such a domain theory and let \mathbb{Z} be an abstract state. The **completion** of \mathbb{Z} relative to \mathbb{C} , denoted $\mathbb{C}^*(\mathbb{Z})$ is the fixed point of applying all domain rules $\mathbb{C}_i \in \mathbb{C}$ to \mathbb{Z} (see Algorithm 18). An abstract state \mathbb{Z} is **legal** iff the completion $\mathbb{C}^*(\mathbb{Z})$ does not contain `false`, otherwise it is **illegal**, i.e. it has no models in the target domain.

A domain theory is an instantiation of the background knowledge theories outlined in Chapter 4. An important difference is that our domain rules can be seen as clauses with multiple heads. In our current setting we mainly use reasoning about constraints on variables used in abstract state descriptions, although the setup allows for more general rules. Without formalizing it, we do assume that the domain rules do not introduce infinite (recursive) loops, and by that, completed states have a finite-length description.

Algorithm 18 A **forward chaining** algorithm to compute an extended state, using domain rules.

Require: \mathbb{Z} is an abstract state and \mathbb{C} is a domain theory.

- 1: **repeat**
 - 2: **repeat**
 - 3: pick a rule $\mathbb{C}_i \in \mathbb{C}$ that is applicable in \mathbb{Z} .
 - 4: compute $\mathbb{Z} := \mathbb{C}(\mathbb{Z})$
 - 5: **until** all rules $\mathbb{C}_i \in \mathbb{C}$ have been tried out
 - 6: **until** no more rules are applicable
 - 7: **return** the extended state $\mathbb{Z} = \mathbb{C}^*(\mathbb{Z})$
-

Algorithm 18 outlines pseudo-code for a *forward chaining* procedure (see also Definition 4.2.15) to compute the completion $\mathbb{C}^*(\mathbb{Z})$ of an abstract state \mathbb{Z} relative to a domain theory \mathbb{C} . A PROLOG implementation is straightforward. Because it is essentially a theorem proving task, there is much room for improvement in the *selection* of the rules and the *order* in which to do this. A more general and efficient way to implement Algorithm 18 is to

use *constraint handling rules* (CHR) Frühwirth (1998). CHR are essentially a committed-choice language consisting of guarded rules with multiple head atoms. CHR define simplification of, and propagation over constraint atoms. Some experiments in Section 6.3.5 make use³⁶ of CHR.

EXAMPLE 6.2.5 ► Two examples of how Algorithm 18 works (we omit most – failed – applications of rules). On the left we complete the state $\text{on}(a, b)$ and on the right the impossible state $\text{on}(a, a)$.

$$\begin{array}{ll}
 \text{on}(a, b) & \Rightarrow_{(\mathbb{L}_4)} \\
 \text{on}(a, b), a \neq b & \Rightarrow_{(\mathbb{L}_1)} \\
 \text{on}(a, b), a \neq b, b \neq a & \Rightarrow \\
 \text{legal state} & \\
 \end{array}
 \qquad
 \begin{array}{ll}
 \text{on}(a, a) & \Rightarrow_{(\mathbb{L}_4)} \\
 \text{on}(a, a), a \neq a & \Rightarrow_{(\mathbb{L}_2)} \\
 \text{on}(a, a), a \neq a, \text{false} & \Rightarrow \\
 \text{illegal state} &
 \end{array}$$

From now on, abstract states will always be completed before being used. Furthermore, action definitions are constrained so that they cannot lead to illegal states. That is, if a state is legal, then applying some action does not lead to a state that is illegal. On the other hand, when computing regression, special care is taken that weakest preconditions are in fact legal abstract states.

More importantly, the notion of θ -subsumption must be extended. As we have said, domain rules can be seen as similar to background knowledge clauses. Buntine (1988) introduced subsumption in the context of a clausal background theory, called *generalized subsumption*. The idea is to first *saturate* the clause (also called *computing the bottom clause*) by a forward-chaining procedure such as the one in Algorithm 18, and to perform a θ -subsumption check on the saturated clauses. Buntine also shows that θ -subsumption is a special case of generalized subsumption when the background knowledge is empty. In practice this means for our approach that subsumption checks are only performed between completed states where the constraint atoms mainly function as additional constraints on the substitutions found.

EXAMPLE 6.2.6 ► Let $Z_1 \equiv \text{on}(X, Y), X \neq Y$ and $Z_2 \equiv \text{on}(a, b)$ be two abstract state descriptions. Clearly, Z_1 will not θ -subsume Z_2 because there is no possibility of finding a substitution θ such that $(X \neq Y)\theta$ occurs in Z_2 . However, the completions of both states are $Z'_1 \equiv \mathbb{C}^*(Z_1) \equiv \text{on}(X, Y), X \neq Y, Y \neq X$ and $Z'_2 \equiv \mathbb{C}^*(Z_2) \equiv \text{on}(a, b), a \neq b, b \neq a$ and now it is easy to see that $Z_2 \preceq_{\theta} Z_1$ in the standard fashion.

And, as a consequence of the new generalized subsumption setting, reductions of abstract states are now possible via subsumption checks on the completed states.

6.2.5 Markov Decision Programs, Value Functions and Policies

The abstract state definitions, reward functions, abstract actions and domain rules together instantiate all components of a FORM (see Definition 4.5.5), thereby inducing a (family of) RMDP(s). We call this particular state description language *Markov decision programs*.

DEFINITION 6.2.7 ► A *Markov Decision Program* $\text{MDPROG } \mathcal{M} = \langle \mathbb{A}, \mathbb{R}, \mathbb{C} \rangle$ is a tuple where $\mathbb{A} = \{\mathbb{A}_1, \dots, \mathbb{A}_n\}$ is a set of abstract action definitions, \mathbb{R} is a reward function and \mathbb{C} is a

³⁶The earlier versions of REBEL used CHR, though in later versions we moved to an all-PROLOG approach and implemented a simple forward-chaining procedure.

domain theory. \mathbb{A} , \mathbb{R} and \mathbb{C} are expressed in a shared language \mathbb{L} .

We will use MDPROGS as our state description language in the next sections, in order to implement an IDP algorithm in first-order domains. Along the lines of Theorem 4.5.1 on FORMs one can prove that every MDPROG induces a (family of) RMDP, possibly of infinite size. Solution components such as value functions and policies are based on decision lists, similar to the reward functions in Definition 6.2.4. In fact, *state value functions* are exactly the same as state reward functions, i.e. a state value function \mathbb{V} is a finite list of *value rules* of the form $\langle \mathbb{S}, v \rangle$ were \mathbb{S} is an abstract state and $v \in \mathbb{R}$. A state's value is the maximum value of all covering rules and value functions are *max-coverings* with default rules.

Abstract state-action value functions are similar to state value functions, but extend these with an action atom of which the arguments are syntactically linked to an abstract state description.

DEFINITION 6.2.8 ► An **abstract state-action value** function \mathbb{Q} is a finite set of *Q-rules* of the form $\langle \mathbb{S}, \mathbb{a}, q \rangle$ were \mathbb{S} is an abstract state, \mathbb{a} is an action atom and $q \in \mathbb{R}$. The **value** of a state-action pair $\langle \mathbb{Z}, \mathbb{a}' \rangle$ is defined as:

$$\mathbb{Q}(\mathbb{Z}, \mathbb{a}') =_{\text{def}} \max\{q_i \mid \langle \mathbb{S}_i, \mathbb{a}_i, q_i \rangle \in \mathbb{Q} \wedge \mathbb{Z} \preceq_{\theta} \mathbb{S}_i \wedge \mathbb{a}' \preceq_{\theta} \mathbb{a}_i\}$$

To any state-action pair \mathbb{z} and \mathbb{a} , \mathbb{Q} assigns the maximum value q of all abstract state action rules subsuming \mathbb{z} , \mathbb{a} . Policies resemble *Q-value functions*, but differ in their semantics.

DEFINITION 6.2.9 ► An **abstract policy** π is a finite, ordered list of policy rules $\langle \mathbb{S}, \mathbb{a} \rangle$, where \mathbb{S} is an abstract state and \mathbb{a} is an abstract action, and $\text{vars}(\mathbb{a}) \subseteq \text{vars}(\mathbb{S})$. The policy action of policy π for a ground state \mathbb{Z} is the following

$$\pi(\mathbb{Z}) = \text{choice} \{ \mathbb{a}\theta \mid \langle \mathbb{S}, \mathbb{a} \rangle \text{ is the first rule in } \pi \text{ such that } \mathbb{Z} \preceq_{\theta} \mathbb{S} \}$$

where *choice* represents a function that chooses among possible substitutions θ .

The *choice-function* is necessary because for a particular ground state multiple – equivalent according to the policy – substitutions of the action's parameters may be generated. We will assume that the *choice-function* is implemented as a uniformly random choice among alternatives. Note that this makes a policy non-deterministic on the ground level, and that a default rule is necessary to ensure the policy is always defined. For both policies and *Q-functions*, we assume that for a state-action pair $\langle \mathbb{S}, \mathbb{a} \rangle$, it holds that $\text{vars}(\mathbb{a}) \subseteq \text{vars}(\mathbb{S})$, to ensure all action parameters are instantiated when using the rule.

Discussion. Markov decision programs provide a simple framework implementing a state description language. Abstract states represent sets of states, and the use of decision lists for value functions and policies induce partitionings of the state (-action) space. We have provided operations for the intersection of sets (e.g. the *GLB*), and for simplification of states (e.g. *REDUCE*). As opposed to propositional state description languages, MDPROGS introduce infinite (or indefinite) domain sizes, action abstraction and parameter sharing between states and actions.

A more general view upon MDPROGS is that they are instantiations of the FORMs in Chapter 4. MDPROGS can be used to specify RMDPs, using a simple logical language. The conjunctive form of states prohibits universal quantification and explicit negation (of

either formulas, or single atoms), and partitionings can only be specified non-declaratively using decision lists, but together with constraint atoms the language is powerful enough for many interesting RMDPs. Related approaches such as the ones by Hölldobler *et al.* (2006) and Wang *et al.* (2007) are based on similar, existentially quantified languages.

In addition, the down-scaled complexity of the logic (as opposed to, for example, expressive logics such as situation calculus) gives us tractable algorithms for theorem proving and the reduction of formulas. Both are very important for expensive algorithms such as first-order DP. A strong argument is given by the fact that, whereas the original symbolic DP approach for situation calculus (Boutilier *et al.*, 2001) was not implemented because theorem proving was too complex in such a full logic, our REBEL approach is computationally feasible, as will be shown in the next sections. Still, all standard issues in FOL reasoning and KR apply to our system, and because of that, IDP in first-order contexts is computationally harder than in the propositional case.

6.3. REBEL: Value Iteration for Markov Decision Programs

The starting point of our approach is a problem domain that we want to solve, assuming it can be modeled as an RMDP (or as a family of RMDPs). The main goal is to find some policy π that is good or optimal for this problem. The first step is to define an MDPROG $\mathcal{M} = \langle \{\mathbb{A}_1, \dots, \mathbb{A}_n\}, \mathbb{R}, \mathbb{C} \rangle$ that models our (family of) problem instance(s). As usual, we set $\mathbb{V}^0 = \mathbb{R}$. Now our relational version of IDP does the following:

given an MDPROG $\mathcal{M} = \langle \mathbb{A}, \mathbb{R}, \mathbb{C} \rangle$, and an initial state value function \mathbb{V}^0 , (e.g. \mathbb{R}), **compute** the sequence of abstract state value functions $\mathbb{V}^1, \mathbb{V}^2, \dots$

That is, we compute a series of value backups, generating a sequence of approximations to the optimal value function for our problem domain, implementing Algorithm 16 using MDPROGS as a state description language. Such a value iteration algorithm for RMDPs iterates over the following steps:

- **Regress** all abstract states \mathbb{V}_i^k in \mathbb{V}^k through all actions \mathbb{A}_j in \mathbb{A} . Each possible **overlap** between an action's effects and an abstract state in \mathbb{V}^k result in a substitution θ that must be backpropagated through the action. Each weakest precondition is augmented with this substitution, because it specifies how the action parameters must be chosen. Regression in our formalism is not a standard operation, and, in addition, the OWA we employ makes things more complex. For these reasons, Section 6.3.1 will describe at length how to compute all possible overlaps, and how to compute all weakest preconditions.
- **Compute \mathbb{Q}^{k+1} using the results from the regression step.** This step is broken into two sub-steps which we will describe in Section 6.3.2.
 1. The regression operator is based on each outcome of actions separately. The first step therefore is to **combine** all the abstract state-action rules that stem from the regression of all different outcomes of one stochastic action. In the current setting, we have to combine state-action pairs, using a GLB construction that takes into account the action variable substitution in the weakest precondition.

2. The second step is to compute a Q -value for the combined abstract state-action pairs, i.e. for the full stochastic actions (see Section 6.3.2). This leaves us with a value function Q^{k+1} that contains actions along with action parameter instantiations.
- Compute V^{k+1} by **maximizing** over Q^{k+1} . Because Q^{k+1} contains instantiated actions, special care has to be taken when maximizing over state-action pairs. In this step we make use of the fact that there is a natural order on the abstract states, i.e. the θ -subsumption order, in order to get a compact representation of the new value function V^{k+1} from the state-action value function Q^{k+1} (see Section 6.3.3).

After converging to the optimal value function V^* an optimal policy can be extracted.

6.3.1 Overlaps, Regression and Weakest Preconditions

The first substantial step in IDP is to compute weakest preconditions needed for the construction of Q^{k+1} , by regressing the value function V^k through the actions in \mathcal{A} . For intensional specifications of actions, we first have to look for the correspondence between the abstract state to be regressed, and the action effects, i.e. we have to compute the *overlap* between these two. The relational approach using Markov decision programs combined with an open world semantics gives more flexibility in characterizing what exactly this overlap is, both semantically and syntactically.

Let us first take an example in a propositional domain where each state is composed of the propositions p_1 , p_2 and p_3 , and let one (deterministic) outcome of some action a be such that it makes p_1 true and it makes p_3 false. Regression is fairly simple here. For any state s where p_1 is true and p_3 is false, regression delivers a state description that is characterized by the action's preconditions, and possibly p_2 's truth value in s . The crucial fact here is that we assume that all the action's effects are matched with parts in the state s . In other words, when the action effects do not mention some aspects of the state, one can assume that they are not changed by that action.

In contrast to the propositional setting, the use of variables in relational descriptions that denote arbitrary objects in the domain, supports a range of new possibilities to define the overlap and to compute regression. Let us take a look at Figure 6.5. In the center the (partial) state description $\mathfrak{s} \equiv (\text{on}(a, b), \text{clear}(a), \text{on}(c, d), \text{clear}(c))$ is depicted. For \mathfrak{s} it is known that there are at least two towers, and the top parts of these towers consist of the blocks a , b , c and d . Note that it is not known what is below these blocks. For example, block d might be resting on another block, or a full stack of hundreds of blocks, or maybe on the floor. The same holds for block b and furthermore, there may be any number of additional stacks in the state that are not mentioned in the description \mathfrak{s} . The four surrounding situations depict some of the possible scenarios from which the application of the first outcome of the action in Example 6.2.2 leads to state \mathfrak{s} . Omitting constraints, the first rule of this action a is defined as

$$a \equiv \text{clear}(X), \text{clear}(Y), \text{on}(X, Z) \xrightarrow{0.9 : \text{move}(X,Y,Z)} \text{on}(X, Y), \text{clear}(X), \text{clear}(Z)$$

Now there are three types of situations, differing in how the overlap is computed between the action's effects and the state \mathfrak{s} .

- The first type is similar to the propositional case, and can be found in Figure 6.5(3). Here, *all* the effects ($\text{on}(X, Y)$, $\text{cl}(X)$, $\text{cl}(Z)$) are matched³⁷ through $\theta = \{X/a, Y/b, Z/c\}$. Block a was on top of c and the action put it onto b.
- The second case is where there is *some* overlap between s and a 's effects. We can see two of these situations in Figure 6.5 (1 and 2). These are the most intuitive situations, given s and a , i.e. the situations in which the action caused either $\text{on}(a, b)$ or $\text{on}(c, d)$. Thus, there is a match with the effect atoms $\text{on}(X, Y)$, $\text{cl}(X)$ but not with $\text{clear}(Z)$. The variable Z stands for the block that X was on *before* moving it onto Y . Thus, in the description of s this *additional* block is not present, but in the regressed state, it is the block where the block to be moved, is resting on.
- The third case is interesting, because there is *no match* between a 's effects and s . One just assumes that all blocks were already in place, and some other block was moved. In this case, three new blocks X , Y and Z are *invented* and the action is assumed to have worked upon them, leaving all blocks mentioned in s in place.

The first case, when translated to the relational case, is the standard mode of regression that is often used in (state-based) regression-based planning systems (see Russell and Norvig, 2003; Brachman and Levesque, 2004). There, regression is employed on complete, ground states. For example, a *goal* state is given as a complete set of atoms describing all facts of the state. In those cases, the overlap between action effects and the state is assumed to be as in the first case above, and each subsequent regression step results in a complete, ground state. The last two cases are typical examples of new possibilities

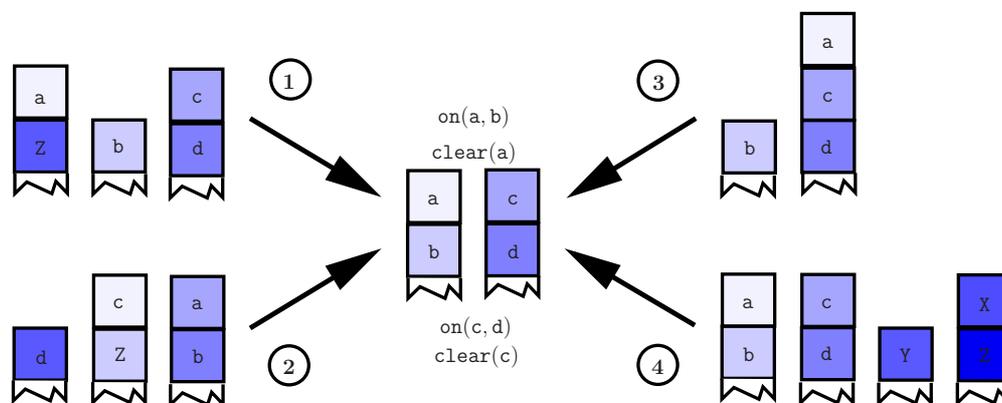


Figure 6.5: Examples of regression in a BLOCKS WORLD. All four configurations around the center state are possible 'pre-states' when regressing the center state through a standard move action. The two left configurations are situations in which either block a is put onto b (1), or block c is put onto d (2). Block Z in both these configurations can be either a block or the floor. In the top right configuration (3) block a was on block c and is now put onto b. The fourth configuration (4) is interesting because it assumes that none of the blocks a, b, c, d is actually moved, but that there are additional blocks which were moved (i.e. blocks X and Y).

for regression in relational domains. Each newly introduced variable may stand for an additional object in the domain, which enlarges the domain, and by that, the state space size. Not many texts deal explicitly with regression in state-based³⁸ representations where

³⁷Note that another possibility is to have $\theta = \{X/c, Y/d, Z/a\}$, resulting in a different pre-state.

³⁸As opposed to *formula-based* regression, e.g. in situation calculus approaches (see also Chapter 4).

there is only a partial overlap with action effects. An interesting, early³⁹ description that is much related to the work in this section, is that by Nilsson (1980, Par. 7.4.) who talks about reversing STRIPS-rules and the employment of constraints to rule out impossible BLOCKS WORLD states. Santos (2000) discusses the use of regression in the context of an open world semantics for *transaction logic*. We have covered the general area of regression approaches in Section 6.1.5.3.

Creating objects on-the-fly is an aspect that clearly highlights new possibilities⁴⁰ of the first-order context. It also highlights the increased complexity of dealing with variable substitutions and constraints. In the experimental section we will explore these matters in more detail and show that it is one of the reasons why *relational* value iteration might not converge. Still, syntactic restrictions can be expressed in the domain theory to make sure there is only a fixed, finite number of objects that must be distinguished in a domain. In fact, in the LOGISTICS WORLD domain we make use of this.

Let us now turn to the regression procedure in REBEL. Let \mathbb{V}^k be the current abstract state value function, say \mathbb{V}^0 , and consider the abstract BLOCKS WORLD action *move* that was defined in Example 6.2.2. For a single Bellman backup, all abstract states $\$$ which lead to a state in \mathbb{V}^0 when taking action *move* have to be computed, i.e. we have to compute all weakest preconditions.

DEFINITION 6.3.1 ▶ Let $\$'$ be an abstract state and let $\text{pre}(a) \xrightarrow{p_i : a} \text{post}_i(a)$ be the i -th action rule for action a . The **weakest precondition** of $\$'$ given the i -th outcome of a , denoted $\text{wp}_i(\$', a)$, is a description of all states $\$$ that lead to $\$'$ when applying a in $\$$ and the i -th outcome is chosen. Each $\langle \$, \theta \rangle \in \text{wp}_i(\$', a)$ consists of an abstract state $\$$ and a substitution θ such that applying $a\theta$ in $\$$ leads to $\$'$.

A weakest precondition of $\$, \text{wp}_i(\$', a)$, covers $\langle \$', a' \rangle$ iff $\$' \preceq_{\theta} \$$ and $a\theta \equiv a'$.

Thus the weakest precondition of an abstract state $\$$ is defined by a *set* of abstract states, each *augmented* with a substitution that denotes the possible ways of substituting the variables in the action definition – and thereby the state occurring in the weakest precondition – such that executing the action leads to $\$'$. We will sometimes make use of the definition of the weakest precondition in an informal way, in the sense that we sometimes speak of a state being in the weakest precondition of some abstract state, given an action. When it is clear from the context that we use a specific substitution θ in $\langle \$, \theta \rangle \in \text{wp}_i(\$', a)$ we omit it.

For example, the first outcome of *move*(a, b, c) can lead from state

$$\$ \equiv (\text{cl}(a), \text{cl}(b), \text{on}(a, c), \text{on}(b, d))$$

(inequality constraints omitted) to the abstract state

$$\$' \equiv \text{on}(a, b).$$

³⁹Interestingly, the author came across this work fairly recently but it remains the most related work on regression and open world semantics for this chapter. Surprisingly, most recent texts do not address the more difficult case of partial matches, newly introduced variables and regression, but only describe regression on ground states. A notable exception is the related relational IDP system FLUCAP by Hölldobler *et al.* (2006) in which the creation of new variables is allowed.

⁴⁰In fact, it is also possible to have a goal $\text{on}(X, Y)$ that merely states that *some block should be on something*. REBEL will compute an infinite value function starting from this.

Thus, we have to compute the *weakest preconditions* for the outcomes of *move* and $\$$. For example, the above state $\$$ lies in the weakest precondition of $\$'$, i.e., $\$ \in \text{wp}_1(\$', \text{move}(X, Y, Z))$ but it does not lie in $\text{wp}_2(\$', \text{move}(X, Y, Z))$.

To compute $\text{wp}_1(\$', \text{move}(X, Y, Z))$ we can assume that we *did* move from $\$$ to $\$'$. Thus, **1)** the preconditions of the action (rule) are fulfilled in $\$$, and **2)** $\$'$ is partially caused by the first outcome of the action. As an illustration of **2)**, consider $\text{on}(a, b)$. If the action did cause $\text{on}(a, b)$, then we have been in abstract state

$$\mathbb{S}_1 \equiv (\text{cl}(a), \text{cl}(b), \text{on}(a, Z), a \neq b, a \neq Z, b \neq Z)$$

and we have moved $X = a$ and $Y = b$. Let us now assume that *move* **did not cause** $\text{on}(a, b)$. Then, we moved X on Y but not a on b . Therefore, we have been in abstract states

$$\mathbb{T} \equiv (\text{cl}(X), \text{cl}(Y), \text{on}(X, Z), \text{on}(a, b), X \neq Y, X \neq Z, Y \neq Z)$$

satisfying that we did not move a on b , i.e., $\text{on}(X, Y) \neq \text{on}(a, b)$, and that we did not move a from b away, i.e., $\text{on}(X, Z) \neq \text{on}(a, b)$. The constraints guarantee that applying *move*(X, Y, Z) in \mathbb{T} preserves $\text{on}(a, b)$. The definition of \mathbb{S} simplifies to \mathbb{S}_2 – \mathbb{S}_5 .

$$\begin{array}{l} \mathbb{S}_2 \equiv (\text{cl}(X), \text{cl}(Y), \text{on}(X, Z), \text{on}(a, b), X \neq Y, X \neq Z, Y \neq Z, \left. \begin{array}{l} X \neq a \\ X \neq a, Z \neq b \\ Y \neq b, X \neq a \\ Y \neq b, Z \neq b \end{array} \right\} \\ \mathbb{S}_3 \equiv (\text{cl}(X), \text{cl}(Y), \text{on}(X, Z), \text{on}(a, b), X \neq Y, X \neq Z, Y \neq Z, \left. \begin{array}{l} X \neq a \\ X \neq a, Z \neq b \\ Y \neq b, X \neq a \\ Y \neq b, Z \neq b \end{array} \right\} \\ \mathbb{S}_4 \equiv (\text{cl}(X), \text{cl}(Y), \text{on}(X, Z), \text{on}(a, b), X \neq Y, X \neq Z, Y \neq Z, \left. \begin{array}{l} X \neq a \\ X \neq a, Z \neq b \\ Y \neq b, X \neq a \\ Y \neq b, Z \neq b \end{array} \right\} \\ \mathbb{S}_5 \equiv (\text{cl}(X), \text{cl}(Y), \text{on}(X, Z), \text{on}(a, b), X \neq Y, X \neq Z, Y \neq Z, \left. \begin{array}{l} X \neq a \\ X \neq a, Z \neq b \\ Y \neq b, X \neq a \\ Y \neq b, Z \neq b \end{array} \right\} \end{array}$$

All \mathbb{S}_i are completed to the same state namely $\mathbb{S}_6 \equiv \text{cl}(A), \text{cl}(B), \text{on}(a, b), \text{on}(A, C)$ where all variables and constants are mutually different.

$$\mathbb{S}_6 \equiv (\text{cl}(A), \text{cl}(B), \text{on}(a, b), \text{on}(A, C))$$

The abstract states $\mathbb{S}_1, \mathbb{S}_6$ together logically define the weakest precondition we are looking for, i.e.

$$\text{wp}_1(\$', \text{move}(X, Y, Z)) \equiv (\mathbb{S}_1 \vee \mathbb{S}_6)$$

So far, we considered a single effect only, namely $\text{on}(a, b)$. In general, however, there can be multiple (combined) effects that are or that are not caused by taking action *move*, cf. Algorithm 19 and Figure 6.5. Consider for example

$$\mathbb{S} \equiv (\text{on}(a, b), \text{on}(c, d))$$

Moving a block onto some other block can have caused either $\text{on}(a, b)$ or $\text{on}(c, d)$, or neither of them. Let us assume that no effect was caused. Then, \mathbb{S}'' is empty and $\mathbb{P} = \text{post}_1(a)$. Therefore, θ is the empty substitution and

$$\mathbb{S} \equiv (\text{on}(a, b), \text{on}(c, d), \text{cl}(X), \text{cl}(Y), \text{on}(X, Z))$$

(inequality constraints omitted) is a possible preimage. However, we know that *move* did not cause $\text{on}(a, b), \text{on}(c, d)$ and therefore, it holds

$$\text{on}(X, Z) \neq \text{on}(a, b) \wedge \text{on}(X, Z) \neq \text{on}(c, d) \wedge \text{on}(X, Y) \neq \text{on}(a, b) \wedge \text{on}(X, Y) \neq \text{on}(c, d)$$

Algorithm 19 Computing the **weakest precondition** $\text{wp}_i(\$, a) = \{\langle \$, \theta \rangle\}$ of abstract state $\$$ and abstract action a (restricted to the i -th, deterministic outcome $\text{post}_i(a)$).

1: $[\text{wp}]$: **Computing Overlaps and the Weakest Precondition**

Require: an abstract state $\$$ '

Require: an action rule $\text{pre}(a) \xrightarrow{p_i : a} \text{post}_i(a)$

Require: a domain theory \mathbb{C}

2: $\text{wp}_i := \emptyset$

3: **for each** $\$'' \subseteq \$'$ % take some aspects of the state

and

$\mathbb{P} \subseteq \text{post}_i(a)$ % take some effects of the action

such that

$\theta = \text{mgu}(\$'', \mathbb{P})$ **exists** % and unify them

or

$\$'' == \emptyset \wedge \mathbb{P} == \text{post}_i(a)$ (i.e., $\theta = \emptyset$) **do**

4: **for all** these mgu's θ **do**

5: $\$:= (\$'\theta \setminus \mathbb{P}\theta) \cup \text{pre}(a)\theta$ % Compute wp state description

6: **for all** pairs $(\mathbb{I}, \mathbb{I}')$ where % Generate all constraints

$\mathbb{I} \in (\$'\theta \setminus \mathbb{P}\theta)$

and

$\mathbb{I}' \in \text{post}_i(a)\theta \cup \text{pre}(a)\theta$ **do**

7: **if** $\text{mgu}(\mathbb{I}, \mathbb{I}')$ **exists then**

8: add $\mathbb{I} \neq \mathbb{I}'$ to $\$$

9: add $\langle \mathbb{C}^*(\$), \theta \rangle$ (if legal) to wp_i

10: **return** wp_i

$\$$ can be simplified for instance to

$$(\$, X \neq a, X \neq c)$$

which is a legal abstract state. The case that the action did cause some effects in the state $\$'$ is covered by the “ $\text{mgu}(\$'', P)$ exists” condition in line 3 and it is treated analogously.

The general strategy that we use for regressing a state $\$'$ through the i th outcome of action A (i.e. $\text{pre}(a) \xrightarrow{p_i : a} \text{post}_i(a)$) is depicted in Algorithm 19, doing the following steps:

- 1 Compute an overlap between a subset \mathbb{P} of $\text{post}_i(a)$, and a subset of the state $\$'$, resulting in an mgu θ . (line 3).
- 2 Compute the pre-state as $\$:= (\$'\theta \setminus \mathbb{P}\theta) \cup \text{pre}(a)\theta$ (line 5).
- 3 Make sure that no literals in the unexplained part $(\$'\theta \setminus \mathbb{P}\theta)$ unify with the action's effects by adding constraints (lines 6–8).
- 4 Make sure that no literals in the unexplained part $(\$'\theta \setminus \mathbb{P}\theta)$ unify with the action's preconditions by adding constraints (lines 6–8).
- 5 Check whether the completed state is legal, and if so, add it to the weakest precondition (line 9).

These steps are repeated for all possible overlaps. Steps 3 and 4 ensure that a forward application of the action results in a state that precisely contains the overlap as a consequence of the action. All other side-effects are ruled out by the added constraints. Because

the conjunctive state language cannot express the logical OR, all possible pre-states must be computed separately and stored. Even though there are many possible overlaps between the state $\$'$ and the action's postconditions, for many of them no MGU can be found. Furthermore, many of the computed pre-states will not be legal, and many legal ones will collapse to the same state before or after completion in step 5.

PROPOSITION 6.3.1 ► The operator wp of Algorithm 19 is sound and complete.

Proof. Soundness is obtained from the fact that when $\langle \$, \theta \rangle \in \text{wp}_i(\$', a)$, applying $a\theta$ in $\$$ leads to $\$'$. Completeness is obtained from the fact that we exhaustively compute all overlaps between $\$'$ and the post-conditions of a . \square

6.3.2 Combination and First-Order Decision-Theoretic Regression

Given the regressed abstract states and the current abstract state value function \mathbb{V}^k , we now compute an abstract state-action value function \mathbb{Q}^{k+1} according to Algorithm 20. To do so, **(A)** we treat each outcome of an action \mathbb{A}_i as though it would be a single action and compute its abstract state action value, cf. line 4. Then, **(B)** we combine the values of all outcomes of an abstract state action value for \mathbb{A}_i , cf. lines 8–13. For the sake of brevity, we will not state constraints in the examples in this section.

DEFINITION 6.3.2 ► Let $\mathbb{Q}_1 = \langle \$, a \rangle$ and $\mathbb{Q}_2 = \langle \$', a' \rangle$ be two Q -rules. The **combination** of \mathbb{Q}_1 and \mathbb{Q}_2 is computed as the GLB of the two clauses $a \leftarrow \$$ and $a' \leftarrow \$'$. That is,

$$\langle \$_g, a_g \rangle := \text{glb}(\langle \$, a \rangle, \langle \$', a' \rangle), \text{ where } \theta \equiv \text{mgu}(a, a'), \text{ and } \$_g \equiv (\$ \theta, \$' \theta)$$

and $a_g \equiv a\theta \equiv a'\theta$. If $\text{mgu}(a, a')$ is undefined, then $\text{glb}(\langle \$, a \rangle, \langle \$', a' \rangle)$ is undefined.

For step⁴¹ **(A)**, consider again the first outcome of move. The weakest precondition was $\text{wp}_1(\text{move}(X, Y, Z), \$') \equiv \$_1 \vee \$_6$. Because $\$_6$ is absorbing, we assign an abstract state action value of $0.9 \cdot 10 = 9.0$ for taking action move, i.e., $\langle \$_6, \text{move}(X, Y, Z), 9.0 \rangle$. The value of $\$_1$, however, is dependent on $\mathbb{V}^k(\$')$, i.e. in our example \mathbb{V}^0 . Assuming a discount factor of 0.9, this yields $0.9 \cdot (\mathbb{R}(\$) + 0.9 \cdot \mathbb{V}^0(\$')) = 0.9 \cdot (0 + 0.9 \cdot 10) = 8.1$, i.e., $\langle \$_1, \text{move}(a, b, Z), 8.1 \rangle$. Doing the same for all other rules in \mathbb{V}^0 results in:

$$\begin{array}{l} \langle a \rangle \\ \langle b \rangle \\ \langle c \rangle \end{array} \left\langle \begin{array}{l} (\text{cl}(X), \text{cl}(Y), \text{on}(a, b), \text{on}(X, Z)) \\ (\text{cl}(a), \text{cl}(b), \text{on}(a, Z)) \\ (\text{cl}(X), \text{cl}(Y), \text{on}(X, Z)) \end{array} \right\rangle \begin{array}{l} , \text{ move}(X, Y, Z) \\ , \text{ move}(a, b, Z) \\ , \text{ move}(X, Y, Z) \end{array} \begin{array}{l} , 9.0 \\ , 8.1 \\ , 0.0 \end{array} \right\rangle$$

For the second outcome of move, step **(A)** leads to:

$$\begin{array}{l} \langle d \rangle \\ \langle e \rangle \\ \langle f \rangle \end{array} \left\langle \begin{array}{l} (\text{cl}(a), \text{cl}(X), \text{on}(a, b)) \\ (\text{cl}(X), \text{cl}(Y), \text{on}(a, b), \text{on}(X, Z)) \\ (\text{cl}(X), \text{cl}(Y), \text{on}(X, Z)) \end{array} \right\rangle \begin{array}{l} , \text{ move}(a, X, b) \\ , \text{ move}(X, Y, Z) \\ , \text{ move}(X, Y, Z) \end{array} \begin{array}{l} , 1.0 \\ , 1.0 \\ , 0.0 \end{array} \right\rangle$$

⁴¹Note that there are several ways to assign values to weakest preconditions. In the IDP framework, we have usually incorporated the rewards after the combination step, ignoring the possibility of absorbing states. In line 4 of Algorithm 20 we do this before combining. Both have their (implementational) advantages. For example, in the current form, it is easier to incorporate action rewards. An alternative is to process all weakest preconditions after the combination step and assign values to absorbing states.

Algorithm 20 Computing a Q -function through decision-theoretic regression. This algorithm computes the Q -function Q^{k+1} for a specific action a using a known state value function V^k . In other words, the value function V^k is regressed through action a to get Q^{k+1} , denoted $Q^{k+1} = \text{FODTR}(V^k, a)$. Note that a denotes the action head where we keep the substitution made by wp_i . Furthermore, note that we use a simple way of incorporating the reward function; we assume that the reward value is constant throughout the pre-states, thereby omitting a full-scale reward function intersection (see the text for explanation).

1: [FODTR] : *Decision-Theoretic Regression for MDPROGS*

Require: An initial Q -value function Q^{k+1}

Require: A state value function V^k

Require: A set of action rules a

Require: A domain theory \mathbb{C} .

Require: A discount factor γ

2: **for each** action rule $\text{pre}(a) \xrightarrow{p_i : a} \text{post}_i(a)$ of a **do**
 3: **for each** $\langle V_i, v \rangle \in V^k$ **do**
 4: $Q_{\text{partial}} := \{ \langle \mathbb{S}, a\theta, \tilde{q} \rangle \mid \langle \mathbb{S}, \theta \rangle \in \text{wp}_i(V, a) \}$
 where $\tilde{q} := \begin{cases} p_i \cdot \mathbb{R}(\mathbb{S}) & : \mathbb{S} \text{ is absorbing} \\ p_i \cdot (\mathbb{R}(\mathbb{S}) + \gamma \cdot v) & : \text{otherwise} \end{cases}$
 5: **if** $Q^{k+1} = \emptyset$ **then**
 6: $Q^{k+1} := Q_{\text{partial}}$
 7: **else**
 8: $Q_{\text{new}}^{k+1} := \emptyset$
 9: **for all pairs** $\langle \mathbb{S}', a', q' \rangle \in Q^{k+1}$ **and**
 $\langle \mathbb{S}'', a'', q'' \rangle \in Q_{\text{partial}}$ **do**
 10: **if** $\langle \mathbb{S}^g, a^g \rangle := \text{glb}(\langle \mathbb{S}', a' \rangle, \langle \mathbb{S}'', a'' \rangle)$ exists **then**
 11: **if** the completed state $\mathbb{S}_c^g := \mathbb{C}^*(\mathbb{S}^g)$ is legal **then**
 12: add $\langle \mathbb{S}_c^g, a^g, q' + q'' \rangle$ to Q_{new}^{k+1}
 13: $Q_{\text{new}}^{k+1} := Q_{\text{new}}^{k+1}$
 14: **return** Q^{k+1}

For step (B), we note that each of these rules describes situations such as *if we are in a state then we can get some value for achieving the i -th outcome of action A* . This information has to be combined to an abstract state action value for A . To do so, we select a rule from $\langle a \rangle - \langle c \rangle$, say $\langle b \rangle$, and a rule from $\langle d \rangle - \langle f \rangle$, say $\langle f \rangle$, and check whether we can be in both abstract states at the same time and whether we can apply the same action. In other words, we compute the *greatest lower bound* (glb) of the logical clauses underlying both value rules. If the glb (where the actions have to unify) exists and it is a legal state, then it is inserted as a new rule, cf. line 11. The value of the new rule is the sum of values of the

combined rules. For $\langle b \rangle$ and $\langle f \rangle$ this yields

$$\begin{aligned} \text{glb} \left(\left\langle \left(\text{cl}(a), \text{cl}(b), \text{on}(a, Z) \right), \text{move}(a, b, Z) \right\rangle, \left\langle \left(\text{cl}(X), \text{cl}(Y), \text{on}(X, Z) \right), \text{move}(X, Y, Z) \right\rangle \right) \\ = \\ \left\langle \left(\text{cl}(a), \text{cl}(b), \text{on}(a, X) \right), \text{move}(a, b, X) \right\rangle \end{aligned}$$

and the new Q -rule $\left\langle \left(\text{cl}(a), \text{cl}(b), \text{on}(a, X) \right), \text{move}(a, b, X), 8.1 \right\rangle$ reflects their combined value. In contrast, $\langle b \rangle$ and $\langle d \rangle$ do not give a new rule. In our BLOCKS WORLD example, Algorithm 20 yields the following abstract state action value function when applied on \mathbb{V}_0 and the actions `move` and `absorb`

$$\begin{array}{l} \langle 1 \rangle \quad \left\langle \left(\text{on}(a, b) \right), \text{absorb}, 10.0 \right\rangle \\ \langle 2 \rangle \quad \left\langle \left(\text{cl}(X), \text{cl}(Y), \text{on}(a, b), \text{on}(X, Z) \right), \text{move}(X, Y, Z), 10.0 \right\rangle \\ \langle 3 \rangle \quad \left\langle \left(\text{cl}(a), \text{cl}(b), \text{on}(a, X) \right), \text{move}(a, b, X), 8.1 \right\rangle \\ \langle 4 \rangle \quad \left\langle \left(\text{cl}(X), \text{cl}(Y), \text{on}(X, Z) \right), \text{move}(X, Y, Z), 0.0 \right\rangle \end{array}$$

Note that we have sorted the Q -rules in descending order only for the sake of readability.

Note that for reasons of efficiency, we use a simple subsumption check to obtain the reward of a state in line 4 of Algorithm 20. We can use this simplified model because in all our experiments in the next section, reward functions are simple. For more general reward functions, we have to *intersect* the reward function with the current state. We have performed some experiments using more general reward functions and these show that this makes REBEL much more computationally expensive.

The general structure of Algorithm 20 is

- 1 For all outcomes of an action a and all states in \mathbb{V}^k weakest preconditions are computed that form the structural basis for \mathbb{Q}^{k+1} for a (line 4).
- 2 Each computed pre-state is augmented with a value, based on the value function \mathbb{V}^k , a discount factor, a partial reward value, and the probability for the particular action outcome (line 4, second part).
- 3 Each abstract state-action pair computed in steps 1 and 2 is combined with the partial Q -function so far, i.e. the computed abstract state-action pairs for other outcomes of the action.

If there are $n > 1$ many outcomes of an action, then the Q -values of the n -th outcome are combined with already combined Q -values of the $n - 1$ previous outcomes. Thus, there are $n - 1$ many combinations per action. This might produce many rules. To overcome this, we adapt Algorithm 21 to compress the Q -function representation by removing redundant rules (see next section); if we are in a state with different currently combined values for compatible actions, then we select only the higher one. This is safe because the higher valued Q -rule subsumes the lower valued one. Therefore, it would have been selected in any case later on.

Variation: Nearly-Deterministic Actions. The combination step is a complex operation for most formalisms. Each iteration of Algorithm 20 takes $|\mathbb{Q}^{k+1}| \cdot |\mathbb{Q}_{\text{partial}}|$ combinations, accompanied by the additional completion and reduction steps. For some classes of domains, we can side-step the combination operator if the actions are *nearly-deterministic*. If we look closely at the probabilistic `move` action in Example 6.2.2, we see that the action either succeeds (with probability p), or fails. Regressing an abstract state \mathbb{V}_i^k through this

Algorithm 21 Maximization of a Q -function. This algorithm computes $\mathbb{V}^{k+1}(\$) = \max_{a \in \mathcal{A}} Q^{k+1}(\$, a)$.

1: [MAXIMIZATION] : *Maximizing a Q -function into a V -function*

Require: a Q -function Q^{k+1}

- 2: initialize $\mathbb{V}^{k+1} := \emptyset$
 - 3: sort Q^{k+1} in decreasing order of Q -values
 - 4: **while** Q^{k+1} not empty **do**
 - 5: remove top element $\langle \$, a, d \rangle$ of Q^{k+1}
 - 6: **if** no other rule $\langle \$', a', d' \rangle$ in Q^{k+1} exists
 such that $\$ \preceq_{\theta} \$'$ **then**
 - 7: add $\langle \$, d \rangle$ to \mathbb{V}^{k+1}
 - 8: remove from Q^{k+1} all rules $\langle \$'', a'', d'' \rangle$
 for which $\$'' \preceq_{\theta} \$$
 - 9: **return** \mathbb{V}^{k+1}
-

action can be done in a simple way. First we regress through the succeeding rule. All states $\$$ in this weakest precondition can be given a full value as

$$p \cdot (\mathbb{R}(\$) + \gamma \cdot \mathbb{V}^k(\mathbb{V}_i^k)) + (1 - p) \cdot (\mathbb{R}(\$) + \gamma \mathbb{V}^k(\$))$$

In other words, we do not combine two pre-states, but instead directly compute a value for $\$$ based on the value of *successfully applying the action and staying in the same state* (which is the abstract state computed during the regression step for the succeeding rule). For this kind of actions, a simplified⁴² version of Algorithm 20 will be much faster. Thus, although we have defined REBEL to work with general, probabilistic actions, specialized versions for these kinds of situations could be devised when applicable.

6.3.3 Maximization: Computing Abstract State Values

The set of Q -rules enables one to compute the next abstract state value function \mathbb{V}^{k+1} . In contrast to the traditional case, Q -rules, i.e., values of abstract state action pairs, can overlap such as Q -rules $\langle 1 \rangle$ and $\langle 2 \rangle$. To compute abstract state values we make use of the fact that $\mathbb{V}^{k+1}(\$) = \max_{a \in \mathcal{A}} Q^{k+1}(\$, a)$. In general, any *value-preserving* transformation can be applied. In this section, we use a simple separate-and-conquer rule learning approach where the rules to learn and the examples to learn from coincide, see the MAXIMIZATION procedure in Algorithm 21. We search for a Q -rule m having a maximal Q -value among Q^{k+1} , separate the covered Q -rules, and recursively conquer the remaining Q -rules by selecting more rules until no Q -rules remain. The main difference is that we select m and add it to \mathbb{V}^{k+1} only if there is no other Q -rule left in Q^{k+1} with the same value whose body subsumes the body of m , cf. line 8. In our running example, we start with rule $\langle 1 \rangle$. Because it is not subsumed by any other rule having the same value, we add $10 \leftarrow \text{on}(a, b)$ to \mathbb{V}^1 and, because it subsumes $\langle 2 \rangle$, we remove $\langle 2 \rangle$ from Q^{k+1} . The remaining highest valued rule is $\langle 3 \rangle$, and we iterate. After completing, this yields the new function \mathbb{V}^1 where we list

⁴²One of the first versions of REBEL was only applicable to these kinds of action definitions, due to a lacking combination operator.

constraints again:

$$\begin{array}{l} \langle (\text{on}(a, b), a \neq b) \quad , \quad 10.0 \rangle \\ \langle (\text{cl}(a), \text{cl}(b), \text{on}(a, X), a \neq b, a \neq X, b \neq X) \quad , \quad 8.1 \rangle \\ \langle (\text{cl}(X), \text{cl}(Y), \text{on}(X, Z), X \neq Y, X \neq Z, Y \neq Z) \quad , \quad 0.0 \rangle \end{array}$$

Note that the transformation depicted in Algorithm 21 does not change the structure of the Q -rules; rules that are inserted in \mathbb{V}^{k+1} are formed by the unchanged states in \mathbb{Q}^{k+1} . The main purpose of maximization is to discard redundant rules and to remove action atoms. In essence, the maximization is built-in by the decision list semantics of \mathbb{Q}^{k+1} .

In addition to maximization, Algorithm 21 can, with minor modifications, be used for at least two other operations:

- **Policy extraction** Extracting a policy from \mathbb{Q}^{k+1} can be done if, instead of gathering states $\$$ with maximum value d (in line 7), one keeps the state $\$$ with accompanying action a found in line 5. Note that will result in a compressed policy representation. An uncompressed representation can be obtained from \mathbb{Q}^{k+1} by just transforming each Q -rule $\langle \$, a, q \rangle$ into a policy rule $\langle \$, a \rangle$; the decision list semantics ensures that the resulting policy always chooses the action with maximum⁴³ Q -value.
- **Simplification** of a Q -function, V -function or policy. In our limited setting of decision lists built from conjunctive states, Algorithm 21 *simplifies* structures by removing redundant rules. Together with the `reduce` operator on states it represents a `simplify` function on value functions and policies.

Further compression, beyond the simple elimination of rules, is possible by providing a richer domain theory, and we will return to these issues briefly after the experiments.

6.3.4 Relational Bellman Backup Operator

Taking the previous algorithms together, we summarize our value iteration algorithm for MDPROGS in Algorithm 22. When we compare it with the structure of the general intentional Bellman backup operator in Equation 6.8 we see that the order of the operations *regression*, *combination*, *maximization* and *merge* has been changed, and interleaved. This is a consequence of taking a concrete state description language that comes with its own opportunities and limitations for implementing the operators making up Equation 6.8. First of all, the *merge* operator is implemented using the reduction operator (see Algorithm 17) and is used at various points during a complete Bellman backup. This ensures that abstract state descriptions stay manageable during the process. In addition, we can employ Algorithm 21 during a Bellman backup to compress (i.e. simplify) Q -functions for individual actions. Furthermore, we *interleave* the combination operator with regression steps; after we have computed a full regression step for one action rule, we combine the results with all structures computed previously for other rules of that same action. The maximization operator maintains its same position as in Equation 6.8, i.e. it maximizes \mathbb{Q}^{k+1} into \mathbb{V}^{k+1} .

⁴³As information about Q -values is lost in the policy representation, the policy is dependent on the existing rule ordering in \mathbb{Q}^{k+1} . For example, let $\mathbb{Q} = \{ \langle \$, a, q \rangle, \langle \$, b, q \rangle \}$. Even though both actions are equally good, the resulting policy π will discard the second rule.

Algorithm 22 Value Iteration using REBEL.**Require:** and $\text{MDPROG} \langle \mathbb{A}, \mathbb{R}, \mathbb{C} \rangle$.

```

1:  $k := 0$ 
2:  $V^0 := \mathbb{R}$ 
3: repeat
4:   for each action  $a \in \mathbb{A}$  do
5:     add  $\text{FODTR}(V^k, a)$  to  $Q^{k+1}$  (using Algorithm 20)
6:    $V^{k+1} := \max_{\mathbb{A}}(Q^{k+1})$  (using Algorithm 21)
7:    $k := k + 1$ 
8: until convergence

```

Formally, Algorithm 22 requires an infinite number of iterations to converge to V^* (e.g. see Section 6.1.1 and also Section 6.3.5). In practice, we stop when the abstract value function changes by only a small amount. Furthermore, in some domains, an optimal policy can be extracted long before value iteration has converged on even the structural level (see Section 6.4.2).

6.3.5 Experiments

In this section we describe a number of experiments with REBEL. Note that the focus of the experiments is on explaining conceptual aspects of our approach. We do not aim at a large-scale validation of efficiency or scaling properties.

- **Logistics Domain Experiment.** This experiment is about the domain described by Boutilier *et al.* (2001). This domain is a simple representative of a large class of problems often used in (probabilistic) planning research. In the experiment we validate REBEL on a concrete instance.
- **BLOCKS WORLD Experiments.** We present three experiments. The results show some of the subtle problems when using the relational state description language of Markov decision programs in IDP. Furthermore, they highlight fundamental differences with propositional approaches.
- **Additional Experiments.** In Section 6.4 we briefly describe some additional results concerning the use of *tabling* and *policy extraction*, and furthermore we mention some aspects of possible extensions concerning universal quantification over goals, topologies and the use of *object identity* instead of constraints.

6.3.5.1 IMPLEMENTATIONAL ISSUES

The initial implementation of REBEL was programmed in the SICSTUS⁴⁴ dialect of PROLOG. Simultaneously it was migrated to a version for the YAP⁴⁵ PROLOG system (v.4.4.4). YAP is a freely available system comparable with SICSTUS, but its execution model proved faster. The initial constraint system was implemented using the *constraint handling rules* library (CHR) Frühwirth (1998). Later on the system was migrated to XSB⁴⁶ PROLOG, that supports tabling.

⁴⁴<http://www.sics.se/sicstus/>

⁴⁵<http://www.dcc.fc.up.pt/~vsc/Yap>

⁴⁶<http://xsb.sourceforge.net/>

The experimental results we report on here were obtained using the YAP implementation. Experiments were run on a 3.1 GHz Linux machine, and the running times were estimated using YAP's built-in `statistics(runtime, ·)`. Furthermore, CHR was used for handling the constraints, though in the text we will describe these using the more general format of Section 6.2.4 for ease of readability.

6.3.5.2 A LOGISTICS DOMAIN

Our first experiment considers a simple logistics domain that is similar to many other domains used in planning research and the *international planning competition*. The characteristics of such domains are that there are certain objects (of certain types) that have to be moved around in a world with different (kind of) locations. Our particular domain was used by the first instance of first-order IDP, the SDP approach by Boutilier *et al.* (2001) in which it was solved semi-automatically.

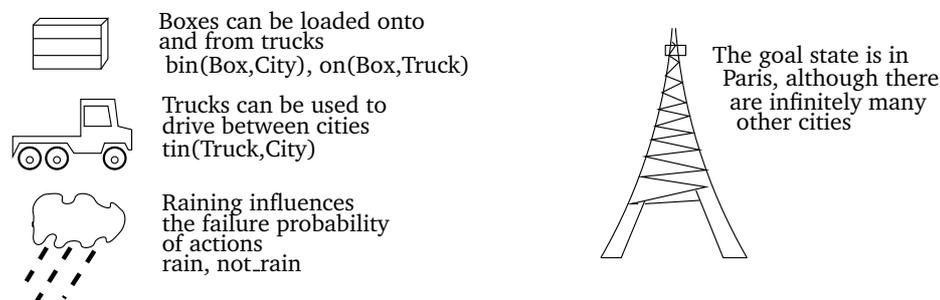


Figure 6.6: *Elements of the logistics domain.* Boxes can be moved around between cities, using trucks that drive between them. Action effects are probabilistic and can be influenced by whether it rains or not.

The domain consists of *cities*, *trucks* and *boxes*. Boxes can be loaded onto and unloaded from trucks, and trucks can be driven between cities. The predicate `on(B, T)` denotes that a box `B` is on the truck `T`, `bin(B, C)` denotes that a box `B` is in some city `C` and `tin(T, C)` denotes that a truck `T` is in city `C`. The actions that can be performed are: `load(B, T)` and `unload(B, T)` specifying how a box `B` can be loaded onto or loaded from a truck `T` and `drive(T, C)` specifying that the truck `T` is driven to city `C`. The actions in this domain have probabilistic effects. The probability of failing a load or unload action, i.e., staying in the current state, depends on whether it rains or not, denoted by `rain`. The action specification (i.e. the set A) is as follows. We omit all rules where an action fails, and furthermore, we omit all constraints, for the sake of brevity.

$$\begin{array}{l}
 \text{on}(\text{Box}, \text{Truck}), \\
 \text{tin}(\text{Truck}, \text{City}), R \quad \xrightarrow{p:\text{unload}(\text{Box}, \text{Truck})} \quad \text{bin}(\text{Box}, \text{City}), \text{tin}(\text{Truck}, \text{City}), R \\
 \\
 \text{bin}(\text{Box}, \text{City}), \\
 \text{tin}(\text{Truck}, \text{City}), R \quad \xrightarrow{p:\text{load}(\text{Box}, \text{Truck})} \quad \text{on}(\text{Box}, \text{Truck}), \text{tin}(\text{Truck}, \text{City}), R \\
 \\
 \text{tin}(\text{Truck}, \text{City1}), \\
 \text{City1} \neq \text{City2} \quad \xrightarrow{1.0:\text{drive}(\text{T}, \text{City2})} \quad \text{tin}(\text{Truck}, \text{City2})
 \end{array}$$

where the transition probability p in the first two action rules is defined as

$$p = \begin{cases} 0.9 & \text{if } R \text{ is } \text{rain} \\ 0.7 & \text{if } R \text{ is } \neg\text{rain} \end{cases}$$

To correctly handle the *explicit negation* we use for `rain`, we provide a constraint that it either rains, or it does not (see below). Note that the above action specification induces nine action rules, i.e. the actions `unload` and `load` can succeed or fail (with probability $(1 - p)$) either when it rains or not (resulting in four rules each) and the action `drive` deterministically succeeds always (resulting in one extra rule). When a `load` or `unload` action fails, the post-state is equal to the pre-state.

The goal in this domain is to get a particular box b in p where p stands for *Paris*, i.e., $\text{bin}(b, p)$ we get a reward of 10. Therefore, the reward function \mathbb{R} is specified as follows:

$$\left\langle \begin{array}{ll} \text{bin}(b, p) & , \quad 10.0 \\ \text{true} & , \quad 0.0 \end{array} \right\rangle$$

Finally, an absorbing action is added as

$$\text{bin}(b, p) \xrightarrow{1.0:\text{absorbing}} \text{bin}(b, p)$$

In the domain theory specification, we can make use of simple *type constraints*. For example, for each $\text{bin}(X, Y)$ atom occurring in a state, we know that the first argument X is of type `box` and the second argument Y is of type `truck`. A third type is `city`. Such constraints can be used internally to restrict the number of matchings between variables, but as shown below, they can be treated just like any other atom. The domain theory \mathbb{C} is specified in the following.

$$\begin{aligned} \text{on}(\text{Box}, \text{Truck}) &\Rightarrow \text{box}(\text{Box}), \text{truck}(\text{Truck}) \\ \text{tin}(\text{Truck}, \text{City}) &\Rightarrow \text{truck}(\text{Truck}), \text{city}(\text{City}) \\ \text{bin}(\text{Box}, \text{City}) &\Rightarrow \text{box}(\text{Box}), \text{city}(\text{City}) \\ \text{box}(X), \text{city}(X) &\Rightarrow \text{false} \\ \text{box}(X), \text{truck}(X) &\Rightarrow \text{false} \\ \text{bin}(\text{Box}, \text{City1}), \text{bin}(\text{Box}, \text{City2}) &\Rightarrow \text{City1} = \text{City2} \\ \text{tin}(\text{Truck}, \text{City1}), \text{tin}(\text{Truck}, \text{City2}) &\Rightarrow \text{City1} = \text{City2} \\ \text{on}(\text{Box}, \text{Truck1}), \text{on}(\text{Box}, \text{Truck2}) &\Rightarrow \text{Truck1} = \text{Truck2} \\ \text{rain}, \neg\text{rain} &\Rightarrow \text{false} \end{aligned}$$

Furthermore, we employ a small number of more general constraints on variables. For example, the first of the following rules expresses the transitivity of the equal-to relation.

$$X = Y, Y = Z \Rightarrow X = Z \qquad X \neq X \Rightarrow \text{false} \qquad X = Y, X \neq Y \Rightarrow \text{false}$$

Note that the constraints we provide here are by no means final. Just like axiom systems (see Chapter 4), various sets of rules can provide the same functionality. The only requirement on a domain theory \mathbb{C} we pose is that the completion of a state should capture the *intended* semantics in our particular domain. Different sets of rules can, however, differ in their effectiveness in computing the completion of a state (i.e. how many steps are needed to complete a state).

REBEL ran for less than 6 seconds to compute the results summarized in Table 6.1 on page 393. After a small number of full IDP iterations, the value function converges. In the table we can see that some of the values are still converging in the last iteration, but differences are marginal. The final value function \mathbb{V}^{10} contains 10 abstract states that make all necessary distinctions in this domain. Consider as an example the underlined value 6.702 in \mathbb{V}^{10} . This is the value of the state

$$\mathbb{V}_7^{10} \equiv (\underline{\text{tin}}(A, B), \text{bin}(b, B), \text{rain})$$

This abstract state description exists in the context of the rules above it (i.e. with higher values) because \mathbb{V}^{10} is a decision list. Therefore, the semantics of \mathbb{V}_7^{10} is that a truck A is in a city B (which is not `paris`, for otherwise we would be in one of the first three states), and the box b is in that same city. In addition, it rains. Now, an optimal policy in this state (taking into account that actions can still fail, resulting in not reaching the goal) is to perform a `load(b, A)` to get the box onto the truck, then do `drive(B, p)` to drive to Paris carrying the box, and then to do `unload(b, A)` to deliver box b in Paris. The abstract state value function applies no matter how many trucks, boxes and cities are present. As concerns the cities, it only matters whether the box (or truck) is in Paris, or *some* other city, and for trucks it matters only that there *is* one truck in our world that we can use to get the box b to Paris. This all could change if we allow, for example, a specification of multiple distinct cities that are connected by roads. In that case, the value function would need to make many more distinctions. Note that whenever an additional city would be introduced, descriptions using it would be removed by the maximization process because a subsumption test would place states containing two *other* cities (i.e. other than Paris) in the same equivalence class with states containing only one *other* city.

6.3.5.3 BLOCKS WORLD EXPERIMENTS

The BLOCKS WORLD has been used as an example throughout this book and, as such, does not need any further introduction. Good reasons for experimenting with this domain are threefold. First, many existing model-free methods use this domain, and it is insightful to see how model-based methods work in this domain. Second, the BLOCKS WORLD is a prototypical, relational domain that is often used in (probabilistic) planning research that is very related to the first-order decision-theoretic planning approaches in this chapter. Finally, it provides a context in which we can show an interesting new insight into the difference between (finite) propositional approaches and (possibly infinite) first-order approaches.

We use the standard `c1/1` and `on/2` predicates to describe abstract states, and additionally use constraint atoms on variables. We employ an open world semantics for our state descriptions, which means that we do not fix (nor completely specify) our domain of objects beforehand. In addition, we use the following rules as our domain theory \mathbb{C} , that is augmented with the general constraints on variables defined in the previous section.

$$\begin{array}{ll} \text{on}(Y, X), \text{on}(Z, X) & \Rightarrow Y = Z \\ \text{on}(X, Y), \text{on}(X, Z) & \Rightarrow Y = Z \\ \text{on}(X, Y), \text{c1}(Z) & \Rightarrow Y \neq Z \\ \text{on}(X, Y) & \Rightarrow X \neq Y \end{array} \quad \begin{array}{ll} \text{on}(X, Y), \text{c1}(Y) & \Rightarrow \text{false} \\ \text{on}(X, Y), \text{on}(U, V) & \Rightarrow X \neq U, Y \neq V \\ \text{on}(X, X) & \Rightarrow \text{false} \end{array}$$

In the following paragraphs we report on three experiments that provide several insights into first-order IDP.

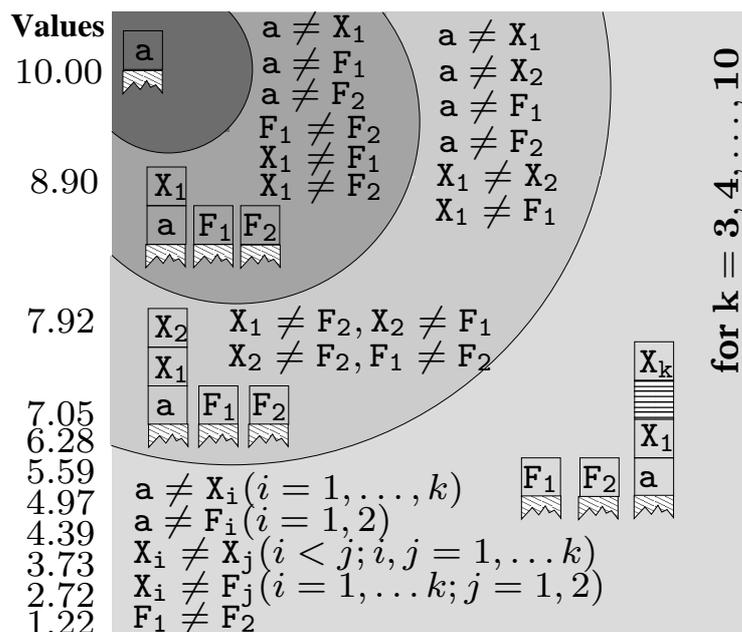


Figure 6.7: Blocks World Experiment I: Abstract state value function for the `clear(a)` goal after 10 iterations. It applies for any number of blocks. Values are rounded to the second digit. F_i can be a block or a floor. States structurally different from the depicted ones get value 0.0.

Blocks World Experiment I. In our first experiment, we consider `clear(a)` as goal in our probabilistic blocks world setting, using the action in Example 6.2.2. The experiment shows that even on simple problems REBEL is not guaranteed to converge (on the structural level). First, we add `clear(a)` $\xrightarrow{1.0: \text{absorbing}}$ `clear(a)` as an action, and we define $\mathbb{R} = \{\langle \text{clear}(a), 10 \rangle, \langle \text{true}, 0.0 \rangle\}$ as the reward function definition. Furthermore, we use a discount factor $\gamma = 0.9$.

We define $g_0 \equiv \text{clear}(a)$ as an (absorbing) goal state with reward 10. All states that can reach g_0 are states in which there is only one block on top of `a` (which we can remove by moving that block to some other place). If we denote them by the abstract state g_1 , then all states that can reach g_0 in two steps, are states that can reach g_1 in one step. In all states in g_2 there are exactly two blocks on top of `a`. In general, after k steps of REBEL, we have computed the optimal k -steps-to-go abstract state value function \mathbb{V}^k and it contains $g_0 \dots g_k$ as abstract states. In each abstract state g_i ($i = 0 \dots k$) there are k blocks on top of `a`. Furthermore, in a deterministic setting it would be the case that $\mathbb{V}^k(g_i) = \gamma^i \cdot 10$, reflecting the number of steps needed to reach g_0 . In the probabilistic setting we consider here, values are somewhat lower because actions can fail.

Figure 6.7 shows the abstract state value function after 10 iterations. It took REBEL roughly 1 minute to iterate ten times. Figure 6.7 highlights that states that are one step further away from the goal get the same value. The value, however, is lower because of the additional block on top of the stack of `a`. The value function clearly shows the one-to-one correspondence between values and *distances* when measured in number of steps.

Now, the interesting, fundamental, result of this experiment is that value iteration in this domain *does not converge*. In contrast to classical value iteration, as well as value iteration in propositional domains, the combined aspects of an open world semantics,

an unknown domain and the capability of introducing new variables during the regression step are responsible for an ever-growing set of value rules. Each iteration of value iteration introduces a new rule \mathbb{V}_m^{k+1} that contains an abstract state description in which there is yet another block Z on top of the stack above the block a . The variable Z does not yet occur in the state descriptions in \mathbb{V}^k and by performing the action $\text{move}(Z, W)$ (where W is again a new variable (which can unify with the floor though) the last state description \mathbb{V}_{m-1}^{k+1} can be reached.

Interestingly, in our previous logistics domain experiment, the final value function \mathbb{V}^* can be approximated using a *finite* set of abstract states that correspond to a finite number of equivalence classes of states. The specifics of the BLOCKS WORLD domain make that one can keep distinguishing between state sets by looking at more blocks. In the logistics domain on the other hand, only few distinctions have to be made, for example between Paris and *any other* city. Note that we can obtain convergence in our BLOCKS WORLD experiment too by restricting the domain of objects, i.e. by adding constraints that put limits on the number of objects allowed to be used in abstract states. Another option is to provide fully ground and complete goal states and to enforce that overlaps between states and action effects in the regression procedure must be total. In this way, the finiteness of the domain is explicitly present in the goal states, but then the only abstraction opportunity is in the action definition and all value functions and policies will be ground. In essence, we are in the standard regression-planning area in this case.

PROPOSITION 6.3.2 ► Abstraction does not guarantee convergence of first-order IDP in infinite domains. Domain characteristics are decisive in whether an infinite number of equivalence classes of states, or equivalently, abstract state descriptions, is needed to represent the optimal value function.

Blocks World Experiment II. We consider the goal $\text{on}(a, b)$ in a deterministic blocks world because it is reported to be a hard problem for model-free relational RL approaches (Džeroski *et al.*, 2001a; Driessens and Ramon, 2003). For instance, Driessens and Ramon (2003) report that on average the learned policies did not reach optimal performance even for 5 blocks (where $|S| = 501$).

Using the same experimental set-up as in our previous experiment, but now with a deterministic move action (taking the action from Example 6.2.2 restricted to the first rule, with probability 1.0), REBEL computed \mathbb{V}^{10} in less than 12 minutes. The abstract value function is partially shown in Figure 6.8. As an example, consider the state depicted in the outer circle. An optimal policy removes the block A from a , and both B and C from b . After that, block a is moved unto b . This all requires 4 actions, resulting in a Q -value of $\gamma^4 \cdot 10 = 6.56$.

Because the move action is deterministic, \mathbb{V}^{10} is optimal for 10 blocks (more than 58 million ground states). The optimal policy can directly be extracted by computing the maximizing Q -rules for each abstract state. In our example, this results in a strategy that removes the top elements from the stacks on top of a and b . However, to compactly represent this strategy, one needs to define the predicate ontop (i.e. captures the concept of being the top block of a stack). In the experiments Driessens and Ramon (2003) reported on, this was always the case. In our current setting of REBEL we have no immediate means to express this kind of predicates, and the best we can do is sum up all states with a different number of blocks on top of a individually. We return to the topic of policy

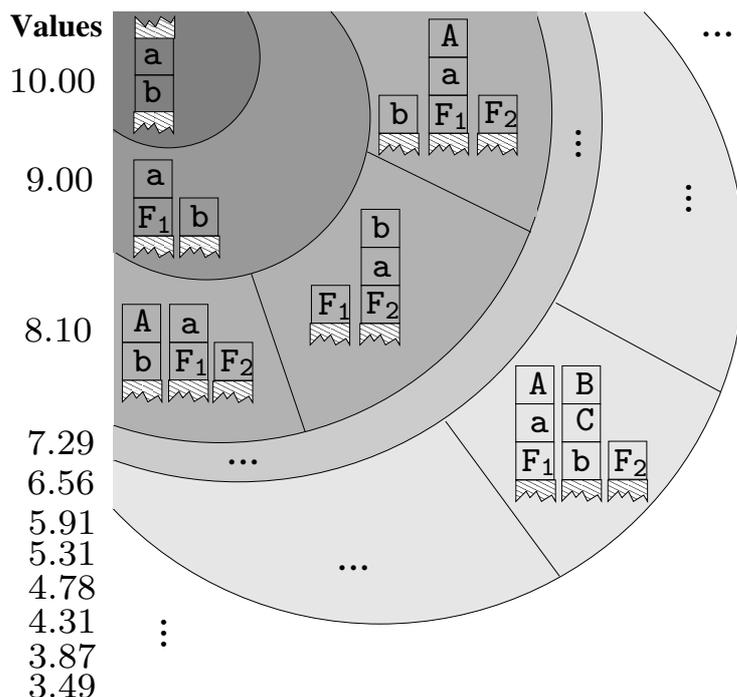


Figure 6.8: Blocks World Experiment II: Parts of the abstract value function for $on(a,b)$ after 10 iterations (values rounded to the second digit). It contains just over a hundred rules. It applies for any number of blocks. We omitted the inequality constraints: All blocks are mutually different. F_i can be a block or a floor block. State more than 10 steps away from the goal get value 0.0.

extraction using additional background knowledge in the next section. The policy based on REBEL is optimal no matter how many blocks there are.

Blocks World Experiment III. Our third BLOCKS WORLD experiment shows some of the complexity and subtleness of IDP in relational domains. We use the same setting as in the previous two experiments, but we extend the probabilistic action definition in Example 6.2.2. In addition to the outcomes of entirely succeeding or failing to move two blocks, we consider a third possible outcome in which the action $move(X,Y)$ does move the intended block X , but accidentally moves it somewhere else than on Y . This new action can be formalized as follows.

$c1(A), c1(B), on(A, C), c1(D),$ $A \neq B, A \neq C, B \neq C,$ $A \neq D, B \neq D, C \neq D$	$\xrightarrow{0.8 : move(A,B,C,D)}$	$on(A, B), c1(A), c1(C), c1(D),$ $A \neq B, A \neq C, B \neq C,$ $A \neq D, B \neq D, C \neq D$	Normal effects
$c1(A), c1(B), on(A, C), c1(D),$ $A \neq B, A \neq C, B \neq C,$ $A \neq D, B \neq D, C \neq D$	$\xrightarrow{0.1 : move(A,B,C,D)}$	$c1(A), c1(B), on(A, C), c1(D),$ $A \neq B, A \neq C, B \neq C,$ $A \neq D, B \neq D, C \neq D.$	The action fails
$c1(A), c1(B), on(A, C), c1(D),$ $A \neq B, A \neq C, B \neq C,$ $A \neq D, B \neq D, C \neq D,$	$\xrightarrow{0.1 : move(A,B,C,D)}$	$on(A, D), c1(A), c1(B), c1(C),$ $A \neq B, A \neq C, B \neq C,$ $A \neq D, B \neq D, C \neq D.$	Accidentally place it elsewhere

The constraints in the action definition ensure that in the third rule the outcome is different from the second. Now the action has two possibilities of failing that differ in their

(structural) results. Below we show the value function computed by REBEL after three iterations.

(1)	\langle	$(\text{on}(a, b), a \neq b)$	10	\rangle
(2)	\langle	$(\text{cl}(a), \text{cl}(b), \text{cl}(A), \text{on}(a, B),$ $a \neq b, a \neq A, a \neq B, b \neq A, b \neq B, A \neq B)$	8.72928	\rangle
(3)	\langle	$(\text{cl}(b), \text{cl}(A), \text{cl}(B), \text{on}(a, C), \text{on}(b, a),$ $a \neq b, a \neq A, a \neq B, a \neq C, b \neq A, b \neq B, b \neq C, A \neq B, A \neq C, B \neq C)$	7.40664	\rangle
(4)	\langle	$(\text{cl}(b), \text{cl}(A), \text{cl}(B), \text{on}(a, C), \text{on}(A, a),$ $a \neq b, a \neq A, a \neq B, a \neq C, b \neq A, b \neq B, b \neq C, A \neq B, A \neq C, B \neq C)$	7.05024	\rangle
(5)	\langle	$(\text{cl}(a), \text{cl}(A), \text{cl}(B), \text{on}(a, C), \text{on}(A, b),$ $a \neq b, a \neq A, a \neq B, a \neq C, b \neq A, b \neq B, b \neq C, A \neq B, A \neq C, B \neq C)$	7.05024	\rangle
(6)	\langle	$(\text{cl}(A), \text{cl}(B), \text{cl}(C), \text{on}(a, D), \text{on}(b, a), \text{on}(A, b),$ $a \neq b, a \neq A, a \neq B, a \neq C, a \neq D, b \neq A, b \neq B, b \neq C, b \neq D,$ $A \neq B, A \neq C, A \neq D, B \neq C, B \neq D, C \neq D)$	4.72392	\rangle
(7)	\langle	$(\text{cl}(b), \text{cl}(A), \text{cl}(B), \text{on}(a, C), \text{on}(b, D), \text{on}(D, a),$ $a \neq b, a \neq A, a \neq B, a \neq C, a \neq D, b \neq A, b \neq B, b \neq C, b \neq D,$ $A \neq B, A \neq C, A \neq D, B \neq C, B \neq D, C \neq D)$	4.19904	\rangle
(8)	\langle	$(\text{cl}(a), \text{cl}(A), \text{cl}(B), \text{on}(a, C), \text{on}(C, b),$ $a \neq b, a \neq A, a \neq B, a \neq C, b \neq A, b \neq B, b \neq C, A \neq B, A \neq C, B \neq C)$	4.19904	\rangle
(9)	\langle	$(\text{cl}(A), \text{cl}(B), \text{cl}(C), \text{on}(a, D), \text{on}(A, b), \text{on}(C, a),$ $a \neq b, a \neq A, a \neq B, a \neq C, a \neq D, b \neq A, b \neq B, b \neq C, b \neq D,$ $A \neq B, A \neq C, A \neq D, B \neq C, B \neq D, C \neq D)$	3.73248	\rangle
(10)	\langle	$(\text{cl}(b), \text{cl}(A), \text{cl}(B), \text{on}(a, C), \text{on}(A, D), \text{on}(D, a),$ $a \neq b, a \neq A, a \neq B, a \neq C, a \neq D, b \neq A, b \neq B, b \neq C, b \neq D,$ $A \neq B, A \neq C, A \neq D, B \neq C, B \neq D, C \neq D)$	3.73248	\rangle
(11)	\langle	$(\text{cl}(a), \text{cl}(A), \text{cl}(B), \text{on}(a, C), \text{on}(A, D), \text{on}(D, b),$ $a \neq b, a \neq A, a \neq B, a \neq C, a \neq D, b \neq A, b \neq B, b \neq C, b \neq D,$ $A \neq B, A \neq C, A \neq D, B \neq C, B \neq D, C \neq D)$	3.73248	\rangle
(12)	\langle	$(\text{cl}(A), \text{cl}(B), \text{cl}(C), \text{on}(A, D), A \neq B, A \neq C, A \neq D, B \neq C, B \neq D, C \neq D)$	0.0	\rangle

Observe the exceptional values for states where $\text{on}(b, a)$ is true (state 3). This is due to the third outcome of the move action. For instance, for the third state in the third line, both the first and the second outcome take the agent to the same state, namely where $(\text{cl}(a), \text{cl}(b))$ holds. In other words, we have a probability of 0.9 of getting to the state in line 2. Due to that we get a slightly higher value than in the states listed e.g. in lines 4 and 5. For the latter ones, the third outcome does not lead to the same states as the first outcome. It might happen that we drop block A on top of b (by accident). Therefore, we only have a probability of 0.8 to reach the state of line 2. This phenomenon occurs later on again, see line 6. In other words, when we try to move block b from a, it does not matter if something goes wrong in the way of the third rule; the end result is that both blocks are free (and we enter state 2). In the fourth and fifth state, however, if something goes wrong, there is a possibility that we move a block that is on a to b (and vice versa).

This experiment shows that by only increasing the problem slightly, the structural form of the value function becomes much more complex. Given that model-free learners have already problems capturing all necessary distinctions in a deterministic world, this extended world will prove even more difficult. This experiment also shows that we might consider an alternative strategy to capture all necessary constraints on variables; it can be seen from the value function that the number of constraints needed grows quickly, thereby increasing computational efforts of most operations present in REBEL.

6.4. Logic Programming meets Dynamic Programming

In Section 6.1.5.3 we have introduced IDP as a generic framework for DP algorithms using structured knowledge representation frameworks. In the previous section we have instantiated this framework with the concrete language of Markov decision programs. The choice for a particular language comes along with a number of instantiations of concepts such as coverage, ordering and reductions. The logical language on which REBEL is based, allows for some extensions using techniques from the field of (inductive) logic programming (see Chapter 4). In general, for first-order logical approaches to IDP one can look at two types of (largely orthogonal) extensions:

- **Dynamic programming:** The order of the backups in DP algorithms, as well as the selection of parts of the state space where backups are to be performed, can be varied as we have discussed in Section 2.5.2. Throughout this chapter, (and Chapter 3), we have seen examples of this. REBEL can be extended in all these directions. An example is the use of search-based techniques in FOVIA (Karabaev and Skvortsova, 2005). Basically, extensions into this direction are about the algorithmic elements and about which parts of the state space are important enough to analyze.
- **Logic programming:** The amount of logical operations inside a first-order logical IDP algorithm offers many opportunities for the insertion of more efficient techniques (e.g. for reduction, overlaps, regression and θ -subsumption) but also for other representations of value functions and policies (e.g. trees, decision lists and partitions, see also Section 4.2.3). Intuitively, this direction is about how we wish to represent the problem and the solution, and how fine-grained we want to reason about it.

Various extensions are possible, such as a more complex state description language, partitions as alternatives for decision lists, or different proof systems. In this section we focus on two such extensions. The first extension is the use of *tabling*, a logic programming technique that can store previously computed derivations and the second is the use of simple inductive generalization to induce policies from a value function. At the end of this section we discuss some additional techniques related to the use of a domain theory.

6.4.1 Tabling

One of the main, computational, bottlenecks of REBEL is the computation of the weakest preconditions in Algorithm 19. Finding all overlaps and regressing these overlaps through the actions, with the addition of constraint handling and reductions, is computationally complex. However, if we closely look at how the value iteration approach in Algorithm 22 works, we can see that many value rules are being *re-computed* in each iteration. For example, all states that were found by regressing V^0 one step back, are being re-computed in each subsequent iteration. For each of these computations, we actually only need a *value* backup, because the *structural* component of the backup (i.e. the abstract state that represents which states have to have their value backed up) has been derived earlier. In other words, if we have computed a structure once, we would like to not do it again.

A general way to avoid redundant structural computations can be found in the technique of *tabling*⁴⁷ developed in the context of logic programming. Tabling amounts to a

⁴⁷See the XSB PROLOG page for references. (<http://xsb.sourceforge.net/>) Tabled logic programming for

memorization of results, which can be seen as trading space for computation time. At a very high level, during computation of a goal to a logic program, each subgoal G is registered in a table the first time it is called, and unique answers to G are added to the table as they are derived. When subsequent calls are made to G , the evaluation ensures that answers to G are read from the table rather than being re-derived using program clauses. For example, the first time query $a(X)$ is used, an SLD-derivation is computed to find a value for X . Any subsequent time that $a(X)$ is used as a subgoal, the answer substitution for X computed the first time, is retrieved instead.

To employ tabling for the weakest precondition computation in REBEL we have two options; one is to use an existing tabling-enabled PROLOG system such as XSB (or to a lesser extent, YAP), and the other is to (independently of a particular PROLOG implementation) store weakest preconditions in an internal database using PROLOG's `assert` and `retract`. In the first approach, tabling can be used for other predicates in the program too, for example when computing subsumption or reductions. We have experimented with both types of tabling, and here we report on some limited experiments to sketch the idea. A full-scale comparison requires additional efforts on the REBEL system itself, and we defer such a comparison to further research.

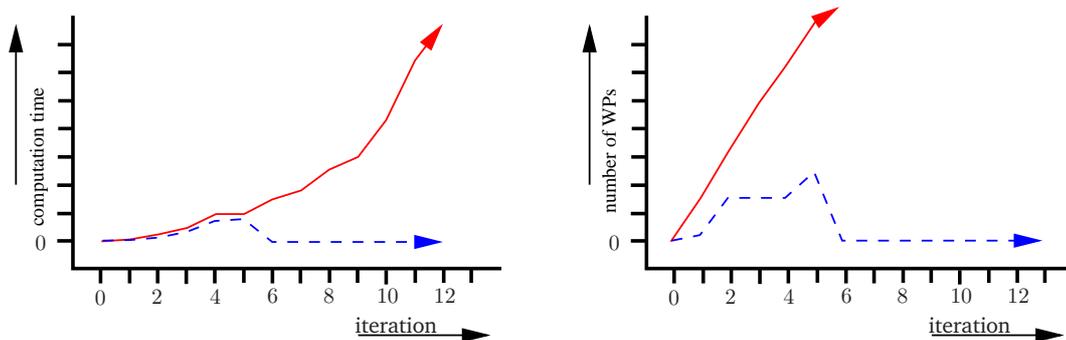


Figure 6.9: Computational benefits of tabling in REBEL. General form of computation graphs for REBEL in the logistics domain. In the left figure, the total computation time is depicted. The upper line denotes the system without tabling. The lower line shows that after a small number of iterations, computation time diminishes because no more structural computations are required. In the right figure, the number of weakest preconditions that are computed is depicted. Without tabling (the upper line), the system keeps computing all weakest preconditions in each iteration. When tabling is used (lower line), after the structural form of the value function is computed, no more weakest preconditions are computed. See (Fischer, 2005) for additional results.

Let us go back to the experiments in the logistics domain in the previous section. We have seen in Table 6.1 that after the fourth iteration, the value function *structure* does not change, but the *values* still do. That means that all weakest precondition structures that are computed after the fourth iteration, are redundant, i.e. they will result in the same value function *structure*, but values are still backed up for these structures. If tabling is used, these structures could be retrieved from memory, without computing them. In Fig-

Horn clause programs was first formalized in the early 1980's. Additionally, several formalisms and systems have been based both on tabled resolution and on magic sets, which can also be seen as a form of tabled logic programming. Although tabling provides a trade-off between time (i.e. of computation) and memory (i.e. storing query results), efficient means are necessary for the search for and retrieval of already computed results. Goals can for example be stored using tries, and the database of such results can grow quickly.

ure 6.9 we have depicted the general results of our experiments (see (Fischer, 2005) for some concrete numbers). For REBEL without tabling, the computation time is rising each iteration. The fact that it rises after the fourth iteration is an artefact, due to memory requirements. The number of weakest preconditions computed remains stable after the fourth iteration; the exact same structures are computed over and over again. For REBEL with tabling we see two predictable improvements. First, computation time falls dramatically after the fourth iteration, because only value updates are performed on a very small abstract state space. Second, the number of weakest preconditions computed after the fourth iteration is zero; the structure of the value function is complete. If we would use tabling in the BLOCKS WORLD examples, things would be slightly different. Computing additional structures would still be needed each iteration, because the structure of the value function is infinite. Still, tabling would help much with the structures already computed.

One fundamental insight we gain from these experiments is the following. First-order IDP computes, up to a certain point, a number of abstract states that represent sets of states in the underlying (R)MDP. After this phase, no new structures are inserted, but the values associated with these structures are still being updated based on value backups that are perfectly aligned with the set-based partitioning induced by the value function. When tabling fully takes over the structural operations, and computation can be fully occupied with updating values, we have just crossed the border between IDP and classical value iteration. In other words, there is a first phase in which the state space is partitioned into abstract states (i.e. sets of states) that *have the same value under the k -steps-to-go optimal policy*, and then a second phase follows in which – in essence – standard value iteration is performed on a new, abstract state space. Related model-minimization approaches (see Section 3.4) focus solely on the first phase, that of computing an abstract state space (based on homomorphisms or stochastic bi-simulations, for example). Both Givan *et al.* (2003) and Boutilier *et al.* (2000a, Thm.1.) talk about the structural convergence of IDP algorithms, but it is our use of tabling that provides us with a very clear border between structural and value convergence. Note that the use of tabling in DP is not novel. Some related approaches (e.g. see Guo and Gupta, 2004, and references in this paper) use logic programming to solve ground instances of MDPs, and these too use tabling to store intermediate results. Novel in our use of tabling in REBEL is the focus on logical abstraction, and the use of tabling for remembering structures, instead of values.

6.4.2 Policy Induction in REBEL

Remember from our first BLOCKS WORLD experiment that we found that REBEL did not converge. This is not due to our system itself, but this result naturally arises from the fact that we tried to compute a value function for an – in principle – infinite state space. In the BLOCKS WORLD in particular, having an infinite number of blocks, paths leading to a goal situation can be infinitely long. These infinite paths require an infinite number of Q -values, and therefore, an infinite number of piecewise-constant value rules.

A general pattern of the growth of value function and policy structures in REBEL is depicted in Figure 6.10. Each iteration will start with a value function V^k of certain size. Using Algorithm 20 a Q -value function Q^{k+1} is computed that is presumably larger, for two reasons. One is that it makes more value distinctions based on action sequences that are one step longer than that in V^k . The other is that Q -functions must necessarily make more distinctions than V -functions. After the maximization step, the resulting V^{k+1} will be

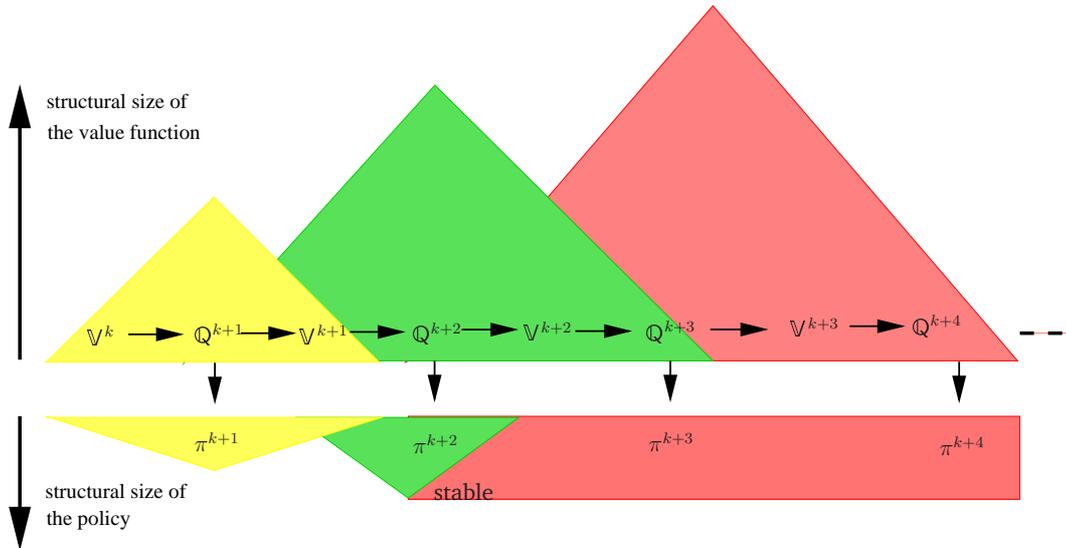


Figure 6.10: Structural growth of value functions and policies.

somewhat smaller, for that same reason. But, we have $|\mathbb{V}^{k+1}| > |\mathbb{V}^k|$, and the continuation of this process will generate bigger and bigger value functions. However, a simple (finite) policy for BLOCKS WORLD experiment I naturally presents itself: if we are in the goal state, we do nothing, and in all other states we identify the stack which holds the block *a*, and remove the top element. The structure needed for this policy can be found already in the first few abstract states of the value function.

This example suggests that we might be able to find a finite policy after a finite number of n iterations of value iteration, even though we would need an infinite number of iterations to compute the value function itself. In the lower half of Figure 6.10 this idea is depicted. The value function keeps on growing with each iteration, though the policy size becomes stable at a certain point. What is needed, are descriptions that capture patterns in the Q -function \mathbb{Q}^{k+1} in a compact, finite form. This can be done by providing additional domain-specific predicate definitions that allow the policy to generalize over state-action pairs with different values. In Chapter 5 we have discussed the P -learning approach by Džeroski *et al.* (2001a) that induces a compact policy representation from ground samples, obtained using a Q -function. Here, we report on a different approach that uses more cautious bottom-up induction using the Q -rules themselves. Because REBEL computes *exact* value functions, we would like to start from these structures as they provide all necessary information. One possibility is to generalize these structures using the least general generalization (see Definition 4.3.7).

DEFINITION 6.4.1 ▶ The **least general generalization** (lgg) (Bergadano and Gunetti, 1995) is defined recursively as follows. The lgg of two terms $f_1(l_1, \dots, l_n)$ and $f_2(m_1, \dots, m_n)$ is a new variable v if $f_1 \neq f_2$ and is defined as $f_1(\text{lgg}(l_1, m_1), \dots, \text{lgg}(l_n, m_n))$ if $f_1 = f_2$. Note that for each recurring occurrence of two terms, the same new variable is used. The lgg of two literals $L_1 = (\neg)p(t_1, \dots, t_n)$ and $L_2 = (\neg)q(s_1, \dots, s_n)$ is undefined if L_1 and L_2 do not have the same predicate symbol and sign; otherwise it is defined as $\text{lgg}(L_1, L_2) = (\neg)p(\text{lgg}(t_1, s_1), \dots, \text{lgg}(t_n, s_n))$. The lgg of two clauses $C_1 = \{l_1, \dots, l_k\}$ and $C_2 = \{m_1, \dots, m_n\}$ is defined as $\text{lgg}(C_1, C_2) = \{\text{lgg}(l, m) \mid l \in C_1 \wedge m \in C_2 \wedge \text{lgg}(l, m) \text{ is defined}\}$.

The length of an lgg of two clauses C_1 and C_2 is at most $|C_1| \times |C_2|$. An lgg usually contains redundant literals, and we can apply a reduction (see Algorithm 17) to get the smallest representation under θ -subsumption. The lgg of two clauses C_1 and C_2 represents a form of *bottom-up generalization*, computing the most specific description C that subsumes both C_1 and C_2 . We employ the same method as for reductions when computing the lgg in the context of background knowledge. That is, we only compute lgg's on saturated abstract states, using the additional predicate definitions in the same way as we have done for constraints. Now consider the following part of the value function for the goal $\text{cl}(a)$ (omitting constraints):

$$\begin{array}{l} \langle \text{cl}(a) \quad , \quad \text{absorb} \quad , \quad 10.0 \rangle \\ \langle \text{cl}(A), \text{on}(A, a), \text{cl}(B) \quad , \quad \text{move}(A, B) \quad , \quad 8.90 \rangle \\ \langle \text{cl}(A), \text{on}(A, B), \text{on}(B, a), \text{cl}(C) \quad , \quad \text{move}(A, C) \quad , \quad 7.92 \rangle \\ \langle \text{cl}(A), \text{on}(A, B), \text{on}(B, C), \text{on}(C, a), \text{cl}(D) \quad , \quad \text{move}(A, D) \quad , \quad 7.05 \rangle \\ \dots \end{array}$$

To generalize over all different-size towers above a , we need the predicate $\text{onTop}(X, Y)$ that expresses (in the form of a domain rule) the fact that block X is the top block of the stack that holds block Y .

$$\begin{array}{l} \text{on}(X, Y), \text{clear}(X) \Rightarrow \text{ontop}(X, Y) \\ \text{ontop}(X, Z), \text{on}(Z, Y) \Rightarrow \text{ontop}(X, Y) \end{array}$$

Using Algorithm 18 state descriptions can be extended with onTop atoms, similar to constraint atoms. We are looking for general patterns in this Q -function that can be exploited by a policy representation using onTop . A general strategy for doing this, is the following:

We can assume that the first rule will appear in a policy representation (there are no other rules with the same action). Now take the second and the third rule. Both rules only differ in the number of blocks on top of a . Using onTop as a domain rule, the second state is extended with $\text{onTop}(A, a)$ and the third with $\text{onTop}(A, B)$ and $\text{onTop}(A, a)$. The following rules contain the two extended states (omitting some of the symmetrical constraints):

$$\begin{array}{l} (C_2) : \text{move}(X, Y) \leftarrow \text{cl}(X), \text{on}(X, a), \text{cl}(Y), X \neq Y, X \neq a, Y \neq a, \underline{\text{onTop}(X, a)} \\ (C_3) : \text{move}(A, C) \leftarrow \text{cl}(A), \text{on}(A, B), \text{on}(B, a), \text{cl}(C), A \neq B, B \neq a, A \neq C, C \neq B, \\ C \neq a, A \neq a, \underline{\text{onTop}(A, B)}, \underline{\text{onTop}(A, a)} \end{array}$$

A generalization of these rules can be obtained using $\text{lgg}(C_2, C_3)$, yielding

$$\begin{array}{l} \text{move}(M, R) \leftarrow \text{cl}(M), \text{cl}(N), \text{on}(M, 0), \text{on}(P, a), \text{cl}(Q), \text{cl}(R), M \neq S, P \neq T, M \neq R, N \neq P, \\ N \neq T, M \neq T, M \neq 0, P \neq a, M \neq U, N \neq U, N \neq 0, N \neq a, \\ M \neq a, Q \neq 0, S \neq a, Q \neq U, R \neq 0, R \neq a, Q \neq a, \text{onTop}(M, 0), \text{onTop}(M, a) \end{array}$$

Note the large amount of atoms and constraints, most of which are redundant and can be removed by a reduction operation. The end result looks like the following rule.

$$\text{move}(M, R) \leftarrow \text{cl}(M), \text{on}(M, 0), \text{on}(P, a), \text{cl}(R), \text{onTop}(M, 0), \text{onTop}(M, a)$$

Because, in this description, $(\text{on}(M, 0), \text{cl}(M))$ entails $\text{onTop}(M, 0)$, we can automatically remove it. Note that this is not the case for $\text{onTop}(M, a)$. The end result is an action rule that subsumes all other rules.

$$\text{move}(M, R) \leftarrow \text{cl}(M), \text{on}(M, 0), \text{on}(P, a), \text{cl}(R), \text{onTop}(M, a)$$

Together with the absorb-rule, this policy is optimal for any BLOCKS WORLD.

The overall strategy to compute a policy from a Q -value function, is to generalize rules using additional language predicates that are useful in the domain. Two challenges here are how to select the rules to be generalized, and how to select additional background predicates that are helpful in the compression of a Q -value function into a policy. So far, we have solved the first problem semi-automatically by hand-picking the rules. A general strategy should search in the space spanned by the rules in the Q -function, similar to how ILP algorithms search in the space of structures. The learning from entailment setting (see Section 4.3.2.1) – on non-ground clauses – would be the right learning setting for this problem. The second problem of finding the right domain predicates is domain-dependent. For our BLOCKS WORLD example onTop is a fairly intuitive candidate. In a domain such as TIC-TAC-TOE (see Figure 3.1) such predicates would be those that describe interesting general patterns on the board that are related to winning, such as FORK and line.

As another example, for the (infinite) value function for the BLOCKS WORLD goal on(a, b) in Figure 6.8 we can perform a similar procedure. Based on a semi-automated procedure in which we hand-pick the rules to be generalized, one possible policy representation is the following (omitting constraints):

$$\left\langle \begin{array}{ll} (\text{on}(a, b)) & , \text{absorb} \\ (\text{cl}(a), \text{cl}(b), \text{on}(a, X), \text{on}(b, Y)) & , \text{move}(a, b) \\ (\text{ontop}(X, a), \text{on}(W, a), \text{cl}(X), \text{on}(a, Y), \text{cl}(Z)) & , \text{move}(X, Z) \\ (\text{ontop}(X, b), \text{on}(W, b), \text{cl}(X), \text{on}(b, Y), \text{cl}(Z)) & , \text{move}(X, Z) \end{array} \right\rangle$$

Note that in this case, the resulting policy is semi-optimal. If the state contains on(b, d), cl(b), on(d, a), then the policy might first remove b from the tower, but a subsequent move might place d on b. Still, based on a random selection function on possible substitutions when applying the action, the policy will arrive at the goal state with high probability. This example illustrates that our generalization procedure is based on induction, and generalization might lose important subtle distinctions.

Summarizing, (finite) policies can sometimes be extracted from infinite value functions by adding extra domain predicates. Such policies may generalize over the entire domain, providing a heuristic *stopping criterion* for value iteration over infinite state spaces, i.e. once the policy structure is stable for a certain amount of iterations, the algorithm can halt. An open research problem is to find an automatic selection of rules to be generalized using lgg constructions, making use of the fact that the value function has a decision list semantics. General inductive procedures could use standard ILP algorithms on the (skolemized) rules, or use a higher-order learner such as ALKEMY (Lloyd, 2003) in a similar way as Gretton and Thiébaux (2004a) did.

6.4.3 Other Extensions and Domain Theories

In addition to experiments concerning tabling and policy induction, there are some open research directions concerning domain theories on which we will elaborate briefly here. So far, we have used domain theories mostly for reasoning with constraints. Still, our formalism is general enough to accommodate more general background knowledge predicates, for example the onTop predicate in the previous section. Conceptually, we can see each state as being comprised of three different parts:

$$\text{state} = \text{basic predicates} + \text{domain facts} + \text{extended facts}$$

The *basic predicates* are the basic constituents of a state, such as `c1/1` and `on/2` in the BLOCKS WORLD. This is the language that actions usually operate on. *Domain facts* and *extended facts* are quite similar, but the difference is that the first is ground and the second is defined using domain rules (e.g. background clauses). Examples of domain facts are facts that are always true. For example, in BLOCKS WORLD the fact `c1(floor)` is always true. It could also be derived using a rule `true ⇒ c1(floor)`. Extended facts are atoms that are derived from basic predicates in the state (and possibly domain facts) using domain rules. Based on a partial state description `on(a, b)` and `c1(a)` one could derive `onTop(a, b)`.

REBEL operates mostly on the level of basic predicates, and in that respect it shares the same problems of the action formalisms we have described in Section 4.4. That is, how to compute action effects in the context of *ramifications*, with the difference now being that REBEL uses *backward* reasoning instead of a standard forward application of action rules. Still, we have *compiled in* some predicates in the form of constraints in the action rules.

One aspect that deserves further research are *domain facts*, because they can represent, for example, *topologies*. A simple extension of our logistics domain would be to include other named states than Paris and define *links* between cities, such that trucks can only travel via these links. This *map* of the world (or, topology) can be assumed fixed, i.e. all facts about city names and their links to other states, are static and true in every state. REBEL supports these topologies in the form of domain rules, and we have performed some experiments in such worlds. However, in practice, states become much more complex, and it would be more efficient to *decouple* all reasoning about the topology and treat them using specialized graph algorithms.

Another aspect is the use of general domain rules, i.e. to derive extended facts. Again, REBEL supports these kinds of facts, though in practice states quickly become unmanageable. As these extended facts correspond to ramifications, in order for REBEL to use them, they must be compiled into the actions, for otherwise the regression operator would not be able to handle the action's effects on their truth value properly. In the policy induction process (see the previous section), such extended predicates such as `onTop` can be of much use to compress the structural value function representation into a policy.

A last interesting direction concerns domain constraints. If we look closely at the BLOCKS WORLD experiment in the last section, we see that in most cases the constraints often ensure that all variables and constants in state descriptions are distinct. Thus, especially in these domains, most constraint handling could be avoided if we have another way to say that all variables and constants mentioned in a description are distinct. One way of doing this is employing the *object identity* (OI) assumption (Khoshafian and Copeland, 1986). OI provides a different semantics to formulas that enforces that all different variables will be bound by different domain objects. An example of the types of expressivity we have in our current language, but that we would lose using OI, is the following. Let $\$ \equiv (\text{on}(a, X), \text{on}(Z, b))$ be a partial state and let there be no constraints. Now $\$$ covers states in which a is on b , because without constraints, it is possible to unify X with b , and Z with a . In contrast, when using OI, this is no longer possible because X and B are supposed to refer to different domain objects. In this case, we explicitly need both possibilities $\$$ and $\$' \equiv \text{on}(a, b)$. We have reimplemented some parts of REBEL to accommodate OI and results indicate that much of the constraint handling can be avoided at the expense of a more limited expressivity. Furthermore, because OI introduces a new semantics of abstract states, most operations operating on states have to be redefined. For example, `lggs`

of states are no longer unique.

In general, most of the new opportunities of using domain theories in first-order IDP are new issues when compared to classical and propositional DP contexts, and many interesting problems are open for further research.

6.5. A Survey of Model-Based Approaches

There are many ways to give an overview of model-based methods for first-order MDPs (see also Sanner and Kersting, 2007, for a brief introduction). The wide range of different languages, domains and techniques prevents us from providing too much detail on the models themselves. Instead, we take IDP and REBEL as described in this chapter as a starting point and give a complete overview of mechanisms of model-based methods. In line with the PIAGET principle defined in Chapter 3 we focus on the structure–parameter and static–dynamic dimensions. Furthermore, we focus on methods that have an abstract model of the RMDP available and use it to compute a solution (either ground or abstract). This excludes for example methods that do assume a model is available, but use model-free RL as a main learning component instead (e.g. Grounds and Kudenko, 2005).

First we survey methods that fully adhere to the *exact* IDP structure. There are currently four such approaches, differing mainly in their state description language. Some extensions within the IDP approach, such as efficient operations are also described. Then, we continue with *approximate* methods within which we can distinguish between *static* and *dynamic* representations. After that, we discuss approaches that use ground solutions and supervised induction of generalized solutions, and the section ends with a brief discussion of additional approaches that go beyond the Markov assumption.

6.5.1 Methods for exact IDP in First-Order Domains

Exact IDP systems are generally required to do the following things: **i)** they assume a fully specified first-order MDP specification, **ii)** they perform model-based solution techniques using complete Bellman backups over logically specified transition functions, **iii)** they are aimed at computing exact value functions, given enough time and space, and **iv)** they perform all computations on an abstract level, *without grounding the problem*. The general outline of first-order IDP methods is presented in Algorithm 23, and it can be seen as a slight generalization of Algorithm 22 for REBEL. Note that we use a general notation in which prob represents a probabilistic distribution over deterministic action choices $\mathbb{A}_j(\vec{X})$, \vec{X} stands for variables occurring in states and actions, and \oplus and \otimes represent summation and product over partitions consisting of formulas. The crucial difference with Algorithm 22 is that we clearly distinguish two separate phases in constructing a new value function \mathbb{V}^{k+1} . The first phase that is tacitly left out in the REBEL approach (for reasons explained below) is the *object-maximization* step (see line 3). For each action type we presumably get rules with different *instantiations* of the same action type, a *parameterized Q-function*. Object-maximization consists of maximizing first over these instantiations. The second phase (line 4) is the standard maximization process over a set of action types to obtain a value function \mathbb{V}^{k+1} from \mathbb{Q}^{k+1} . Note that when rewards are associated with states (as we have done throughout this chapter) we can replace lines 2 and 3 by

$$\mathbb{T}_{\mathbb{V}^k}^{\mathbb{A}(\vec{X})} = \oplus_j (\text{prob}(\mathbb{A}(\vec{X})) \otimes \text{Regr}(\mathbb{V}^k, \mathbb{A}_j(\vec{X}))) \quad \text{and} \quad \mathbb{Q}_{\mathbb{V}^k}^{\mathbb{A}} = \mathbb{R} \oplus \gamma \otimes \text{obj-max}(\mathbb{T}_{\mathbb{V}^k}^{\mathbb{A}(\vec{X})})$$

Algorithm 23 First-Order Dynamic Programming in general form. It computes V^{k+1} via the Q -rules computed from V^k for all actions.

Require: an abstract value function V^n

- 1: **for each** action type $A(\vec{X})$ **do**
 - 2: compute $Q_{V^k}^{A(\vec{X})} = R \oplus [\gamma \otimes \oplus_j (\text{prob}(A(\vec{X})) \otimes \text{Regr}(V^k, A_j(\vec{X})))]$
 - 3: $Q_{V^k}^A = \text{obj-max}(Q_{V^k}^{A(\vec{X})})$
 - 4: $V^{k+1} = \max_A Q_{V^k}^A$
-

Currently, four published version of exact, first-order IDP algorithms have appeared in the literature. The first method is the *symbolic dynamic programming* (SDP) approach (Boutilier *et al.*, 2001; Boutilier, 2001) based on the *situation calculus* (SC) language (Reiter, 2001; McCarthy, 1963). The second approach is based on the *fluent calculus* (Thielscher, 1998), and is denoted here as FOVI⁴⁸ (Großmann *et al.*, 2002; Hölldobler and Skvortsova, 2004a; Karabaev and Skvortsova, 2004). The third approach is our REBEL approach (Kersting *et al.*, 2004; van Otterlo *et al.*, 2004; Fischer, 2005) which has been one of the main topics of this chapter. The most recent approach is based on *first-order decision diagrams* (see Groote and Tveretina, 2003), and will be called FODD here (Joshi *et al.*, 2006; Wang *et al.*, 2007; Wang, 2007; Wang and Khardon, 2007; Wang *et al.*, 2008a).

Representation. REBEL and FODD are based on simple relational formalisms that rely on implicit negation. REBEL supports limited negation on variable bindings through constraints which could be used as normal atoms in FODD too. FOVI is slightly more general, in that it supports negated fluents in the state description. SDP represents the other end of the spectrum where states are represented by general FOL formulas. This makes it possible to generate and maintain *explicit* partitions, whereas the other three methods use implicit representations, either by decision diagrams or decision lists. In REBEL, each partition is represented *implicitly* by the negation of all rules above it, and explicitly in the conjunction in the rule. FODD lifts some of the ideas of using propositional ADDs for MDPs (Hoey *et al.*, 2000; St-Aubin *et al.*, 2001, see also Section 3.5) to the relational case. First-order decision diagrams are similar to first-order decision trees, with as main difference that branches can converge. FODD uses a *multiple path semantics* representing a *max*-covering as the maximum value of all different paths (i.e. substitutions) to leaf nodes. This type of semantics is shared by REBEL and FOVI that use subsumption of rules in the context of decision lists. Transition functions in all formalisms are based on the underlying logic; probabilistic STRIPS rules for REBEL, decision diagrams in FODD, and both SDP and FOVI use their underlying fluent and situation calculus action specifications. Figure 6.11 shows some examples of representations used in SDP and FODD. The representation used in FOVI is similar to that of REBEL; the state language is restricted to existentially quantified (possibly negated) fluent terms. For example, an abstract state in their representation is the following

$$Z \equiv \left(\underbrace{\text{on}(X, a) \circ \text{on}(a, \text{table})}_{\text{positive}}, \underbrace{\{\text{on}(Y, X), \text{holding}(Z)\}}_{\text{negative}} \right)$$

⁴⁸We choose FOVI as the general name here, although the literature on this method contains quite a few names, based on whether the concrete implementation is referred to, and various components of the method. Names include FCPLANNER, LIFT-UP, FLUCAP and FOVIA.

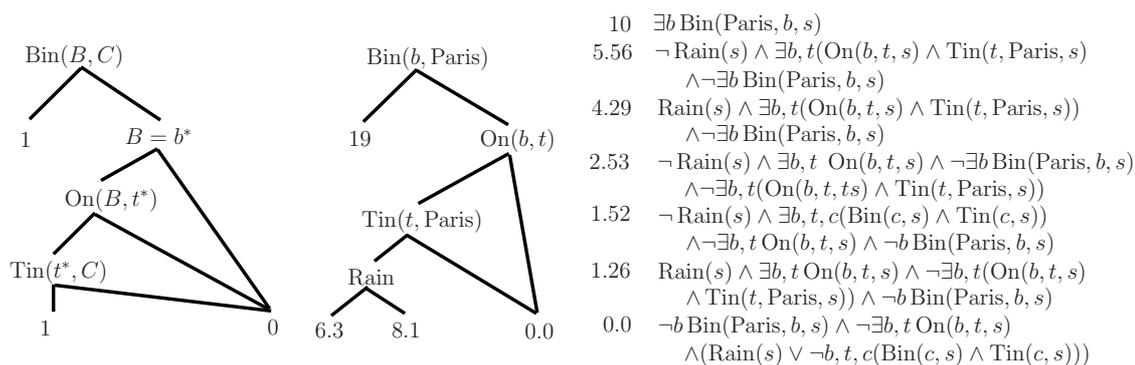


Figure 6.11: Examples of structures in exact, first-order IDP. Both are concrete examples using the logistics domain described in our experimental section. The left two figures are first-order decision diagrams taken from (Wang et al., 2007). On the far left, the transition function (or, truth value diagram) is depicted for $\text{Bin}(B, C)$ under action choice $\text{unload}(b^*, t^*)$. The right diagram depicts the value function \mathbb{V}^1 , which turns out to be equivalent to $\mathbb{Q}_{\text{unload}}^1$. The formulas on the right represent the final value partition for the logistics domain computed in (Boutilier et al., 2001).

and its meaning is that X is on a which itself is on the `table`, and furthermore, there is *no* block Y on X , and also the gripper is *not* holding any block. The special function symbol \circ can intuitively here be read as *AND*, but it is used to reason with abstract states as *multisets*. Note how exact, but also how complex, a simple value function in SDP becomes. The decision diagrams used in FODD are highly compact, but strictly less expressive than SDP state descriptions.

Operations. General reasoning patterns in the four methods are all instantiations of general reasoning by logical entailment. In REBEL we employ subsumption to do most of the reasoning. FODD employs efficient datastructures (i.e. decision diagrams) and all reasoning is done in terms of these diagrams, such as addition and simplification. SDP uses general, first-order reasoning in the situation calculus. The FOVI method relies on the fluent calculus, which is a first-order logic program with SLDE-resolution. Subsumption in FOVI is used separately on the positive fluents in a state, and the negative ones.

Regression is built-in as a main reasoning method in the SDP language (see Section 4.4.2.2). In FOVI this type of inference is based on the AC1 equational theory for the special function symbol \circ , working on abstract states in terms of multisets. In Section 6.3.1 we have described regression in REBEL, where we must enumerate all possible matches between a subset of a conjunctive goal (in the value function) and the action effects and reason about them separately. Although much of this work must be done in other systems too, in Algorithm 19 we make all separate steps explicit. The FODD system performs regression through each deterministic action choice by operations that combine the diagram for \mathbb{V}^k with diagrams for the actions. REBEL and FOVI have both been defined for an open world semantics, i.e. introducing new state variables when necessary (see Section 6.3.1).

A general pattern in reasoning and regression procedures is that each state description language comes long with reasoning methods that best suit that logic. The main difficulties lie in how to compute and represent a complete first-order decision-theoretic regression of the current value function \mathbb{V}^k through all probabilistic actions, and how to combine separate, deterministic actions. FOVI combines the regression step and the combination step

into one reasoning step through a special causality predicate *causes/3*, that is invoked on all deterministic outcomes of an action type at once (see Skvortsova, 2003, for examples.). The other three methods separate these steps and compute weakest preconditions for each action outcome separately. Conjoining all weakest preconditions for separate outcomes of one action type is logically simple, yet computationally expensive, because many combinations must be considered. The real difficulty lies in the reduction of the state descriptions that appear in $Q_{\mathbb{V}_n}^A(\vec{x})$. In REBEL we employ simplification on single rules, whereas FODD uses simplification on diagrams. An interesting feature of the implicit state partitions in FOVI, REBEL and FODD is that they do not require explicit *object-maximization*. In SDP the Q -function represents a partition, such that it must explicitly maximize over action instantiations in $Q_{\mathbb{V}_n}^A(\vec{x})$ to obtain $Q_{\mathbb{V}_n}^A$, because these action instantiations can overlap. In SDP object-maximization is done by first sorting the partition in decreasing order (on the values) and including the negated conditions for the first n partitions in the partition formula for the $(n + 1)$ th partition, explicitly enforcing that a partition can only be satisfied if no higher-valued partitions can be satisfied. Note that this leads to complicated formulas that must be simplified before further computations can be made. FOVI, REBEL and FODD on the other hand, get object-maximization for free by allowing overlapping partitions in their value function representations, where the correct values are ensured by their semantics (i.e. max-coverings).

Finally, the maximization process in FOVI, REBEL and FODD is simpler than in SDP. Again, in SDP the partition is sorted, and a new partition for \mathbb{V}^{k+1} is enforced by including all negated formulas of higher-valued partitions in the lower-valued partitions. In the FODD approach, diagrams for the Q -function can be added (and reduced and maximized) to form $\max_A Q_{\mathbb{V}_k}^A$. Both REBEL and FOVI only need to gather all rules in the Q -function and sort them, and remove redundant rules (see Algorithm 21 and (Hölldobler and Skvortsova, 2004b)) and the decision-list semantics gives an implicit maximization.

All four methods perform IDP in first-order domains, *without grounding the domain*, thereby computing solutions directly on an abstract level. The core reasoning procedure is a first-order version of *decision-theoretic regression* (FODTR) (Boutilier *et al.*, 1999). On a slightly more general level, FODTR can be seen as a means to perform *first-order* reasoning over decision-theoretic values, i.e. as a kind of decision-theoretic logic. One can say that all four methods *deduce* optimal utilities of states (possibly in infinite state spaces) through FODTR, using the action definitions and domain theory as a set of axioms. From these four methods, only two have been accompanied by concrete implementations. REBEL and FOVI have been tested on the logistics domain and BLOCKS WORLDS (see Section 6.3.5), and in addition, (extensions of) the FOVI approach have entered the *international planning competition* (IPC) in 2004. The first approach, SDP has not yet been shown to be implemented. The main bottleneck appears to be the first-order theorem proving that is needed to simplify the partitions. Very recently, FODD has been implemented and shown to be comparable in speed to the REBEL approach described in this chapter (on the logistics domain). The resulting FODD-PLANNER (Joshi and Kharon, 2008) also handles background knowledge predicates following the idea we have described in Section 6.4.3.

In general, the more complex the representation language, the more complex these reduction operations are. Still, expressive logics such as used in SDP are desirable. For one thing, expressive languages support a rigorous formalization of complex problems.

Also, such languages support general, explicit partitions where each formula can be considered independently, with possible extensions towards more *local* FODTR such as in the propositional case (Dearden, 2001). Furthermore, partitions can be exploited in algorithms because states can be considered in isolation (in contrast to e.g. states in a decision list). Second, REBEL, FOVI and FODD are limited to problems that can be specified by *existential* conjunctions. A simple goal in the BLOCKS WORLD in which *all blocks* are on the floor, cannot be expressed in these systems in a natural way. For this, one would need to be able to express the goal as $\text{unstack} \equiv \forall X \text{on}(X, \text{floor})$. For finite domains, one could for example specify the goal state in ground form. But, if the goal is changed to *stack*, i.e. *all states in which all blocks form one big tower*, there are $n!$ ground goal states when there are n blocks. *Universally quantified goals* such as *unstack* will always be difficult in first-order IDP systems, because they pose problems for domain-independent solutions on an abstract level. In contrast, model-free methods (e.g. Džeroski *et al.*, 2001a, and see further Chapter 5) have no real problems when learning with universally quantified goals such as *stack* or *unstack* because they do not reason about the goal description, but only use it as a *test* on whether the goal state has been reached.

6.5.1.1 EXTENSIONS AND EFFICIENT OPERATIONS

Without fundamentally modifying the first-order IDP systems we have described, a number of *logical* operations in these systems can be made more efficient. When various logical operations have been identified, research into efficient implementations can be done orthogonally to the methods in which they are used. Also, some extensions can be defined as minor extensions of the methods themselves. In REBEL we have described the use of *tabling* and novel methods for policy induction (see Section 6.4). Other extensions are concerned with efficient datastructures such as *tries* for increased efficiency in overlap computations, and in-between compression using Algorithm 21 as we have described. One additional extension is the exploration of *asynchronous* updates in the REAVER algorithm (see Fischer, 2005).

Exact FOVI was extended with search-based exploration of the state space (see later in this section), and it was extended with efficient subsumption tests. In Section 6.3 we have highlighted the complexity of subsumption checking, and its prominent place in various parts of our algorithm. FOVI shares many of these matters with REBEL and efficient methods can improve first-order VI very much. Several ways of computing domain-*dependent* and domain-*independent* subsumption, based on graph-algorithms, have been studied (Karabaev *et al.*, 2006; Skvortsova, 2006a,b,c) and their incorporation in the FLUCAP system has improved the overall solution to first-order MDPs.

Wang and Khardon (2007) discuss the use of *policy iteration* for RMDPs. Although this extension is relatively straightforward in the context of first-order IDP, their analysis highlights some of the caveats of restricted languages. What is needed for the policy evaluation step is the regression *through a policy* (or the intersection with a policy structure, as we have mentioned in this chapter). Wang and Khardon show that when using restricted languages that use *implicit* state partitions, the policy evaluation step automatically entails a maximization step, which also entails that the value function that is computed, is actually the value function of another (better) policy. This is due to a re-arranging of the value (and policy) rules based on the values of these rules (e.g. as in Algorithm 21 when used for compressing a Q -function or policy). More general state languages (e.g. as in

SDP) that support universal quantification can represent *explicit* partitions and are not affected by this. All three systems REBEL, FOVI and FODD have these restrictions. Still, Wang and Kharon prove that their *relational modified policy iteration* (RMPI) algorithm converges, even dominating the iterates from VI. However the question is still open on which to prefer; RMPI converges faster, at the expensive of an increased computational complexity.

A recent extension of SDP was proposed by Sanner and Boutilier (2007), making several contributions in finding more types of *structure* in first-order MDPs and exploiting this structure in efficient algorithms. The so-called *factored* FOMDP model supports the specification of *factored actions*, making independence of sub-actions explicit which can be exploited in more efficient backup operations. Furthermore, *additive reward models* are added that scale with the domain size. A complete derivation of an SDP algorithm is given, but because of the expected blowup of the value function and the computational complexity of theorem proving, no experiments for exact value functions are reported. Instead, a linear value function approximation is used and we discuss it briefly in the next section. Finally, the SDP representation has been extended towards FOADDs and trees (see Sanner, 2006b), but no results have been reported yet.

A last type of extension is the support for *universal quantification* over goals (see previous section). Several authors (e.g. Yoon *et al.*, 2002; Gretton and Thiébaux, 2004a) have mentioned that this type of feature is desirable (and even one of the most pressing issues in the practical solution of first-order MDPs), but most current systems do not support these kinds of goals. Universally quantified goals depend on a domain instantiation and thus usually prevent domain-independent solutions. Furthermore, the *range* of values can scale with the number of domain objects. One solution was provided by Sanner and Boutilier (2006) using a *decompositional approach*. Intuitively, when a (positive) goal-based reward is represented as $\forall Xg(X)$ (and zero otherwise), one could decompose it into a set of ground goals $\{g(X_1), \dots, g(X_n)\}$ for all possible X_j in the ground domain. If all ground goals are true, then $\forall Xg(X)$ is true. However, general FOMDP algorithms work without knowledge of this domain, and thus the set of ground goals is unknown. The solution chosen by Sanner and Boutilier is to use a *generic* ground goal $\forall Xg(X^*)$ for a *generic* object vector X^* . A generic solution (i.e. an optimal Q -function) is *instantiated* with ground goals when faced with a concrete domain. This type of approach only works for universally quantified goals (not mixed with existentially quantified predicates) and only in some domains. Still, it represents one practical solution, although it is an open problem how to tackle the general problem of complex reward functions.

6.5.2 Approximate Model-Based Methods for First-Order MDPs

Exact value functions are very desirable because they allow for exact solutions and optimal policies to be computed. However, given the high computational complexity of exact methods, several authors have explored ways to approximate either the problem or the solution, in order to make algorithms more tractable, and to scale up to larger problems where exact methods break down. Basically, there are three ways to approximate. The first is to define a fixed abstraction level that is sufficient for representing the problem and estimate parameters on this level. This gets rid of various logical reasoning problems and provides opportunities for rigorous performance bounds (Section 6.5.2.1). A second opportunity is to use an approximate representation of the value function, but change it over the course

of learning. Such structure-changing algorithms make algorithms more complex, but they can adapt their representation to the problem at hand (Section 6.5.2.2). A third class of models first uses exact solutions on a ground level of states and actions, and uses generalization methods to obtain (approximated) abstract solutions (Section 6.5.2.3). Whereas the first type of systems belongs to PIAGET-1 (or PIAGET-2), the second two classes are PIAGET-3, combining both structure and parameter learning.

6.5.2.1 STATIC REPRESENTATIONS

Static approaches generate, or specify, an abstraction level once, and then do parameter estimation, in line with PIAGET-1 or PIAGET-2.

The approach by Guestrin (2003) (see also Guestrin *et al.* (2003a)) improves on the representation of value functions by using the *probabilistic relational model* (PRM) framework (Getoor *et al.*, 2001). A PRM models a probability distribution over a first-order representation. A crucial assumption of the approach is that the relations between objects are *static*. While this may not look unrealistic for domains such as the subtask of the computer game FREECRAFT on which the approach is tested, the majority of tasks such as used in planning systems or in any of the other relational RL systems – including BLOCKS WORLD – involves relations that change over time. In this limited setting, the global value function is assumed to be decomposed additively into local value functions for each object, so-called *class-based value functions*. It is assumed that all objects have a certain *class* and the local value functions can vary from class to class. The local, class-based value functions are assumed to only depend on some of the properties of those objects and objects that might be directly related to them. The method uses an efficient linear programming procedure to find appropriate weights for the linear combination of the local object properties. If all assumptions – including the ones described about the additive nature of the value functions – hold, the method can be guaranteed to approximate the true value function within some bound. The work provides bounds for the generalization to worlds with different numbers of domain objects and presents good empirical results in large domains. It shares with the other modeling approaches the difficulties of the modeling involved, in this case the definition of basis functions that can be combined into a global value function. Recently, Diuk *et al.* (2008) used a representation similar in spirit, for *object-oriented* RL.

The *first-order approximate linear programming* technique (FOALP) (Sanner and Boutilier, 2005) extends the SDP approach, transforming it into an *approximate* value iteration (AVI) algorithm (Schuurmans and Patrascu, 2001, see also Section 3.5.2). Instead of exactly representing the exact and complete value function, which can be large and fine-grained, and because of that hard to compute, the authors use a fixed set of *basis functions* (see also Definition 4.5.10). That is, a value function can be represented as a *weighted sum of k first-order basis functions*, denoted $\text{bCase}_i(s)$, each containing a *small* number of formulae that provide a first-order abstraction (i.e. partition) of the state space.

$$\text{vCase}(s) = \bigoplus_{i=1}^k w_i \cdot \text{bCase}_i(s)$$

An example of a value function is the following:

$$\text{vCase}(s) = w_1 \cdot \text{case} \left[\left(\text{clear}(a), \text{on}(a, b) \right), 10; \neg, 0 \right] \\ + w_2 \cdot \text{case} \left[\left(\exists X, Y \text{on}(X, Y), \text{on}(Y, a), X \neq b, Y \neq b \right), 7; \neg, 0 \right]$$

Note that the case-statements act as binary features. The second part of both partitions

represents the negation of the first part. The backup of a linear combination of such basis functions is simply the linear combination of the FODTR of each basis function:

$$B^{A(\vec{x})}(\oplus_i w_i \text{bCase}_i(s)) = \text{rCase}(s, a) \oplus (\oplus_i w_i \text{FODTR}(\text{bCase}_i(s), A(\vec{x})))$$

Unlike exact solutions where value functions can grow exponentially in size (see this chapter) and where much effort goes into logical simplification of formulas, this feature-based approach must only look for good weights for the case statements. Each step in AVI does not generate new logical structures, but instead a set of *constraints* on the values of the basis function contribution to the global value function are generated. The constraint generation problem is more complex in the FOL case than in the propositional case because of a possibly infinite domain. The problem when just generalizing the propositional LP formulation to the first-order setting is that it is ill-defined: it would sum over infinitely many situations. The remedy is to sum over each case partition (equivalence classes of states). This constraint set can be efficiently solved using LP techniques and the approach was tested in a small elevator dispatch problem. An important point is that an error bound can be computed on the error that is introduced due to approximations.

Related to FOALP, the *first-order approximate policy iteration* algorithm (FOAPI) (Sanner and Boutilier, 2006) is a first-order generalization of approximate policy iteration for factored MDPs (e.g. see Guestrin *et al.*, 2003b). It uses the same basis function decomposition of value functions as the FOALP approach, and in addition, an explicit policy representation. Assuming there are m parameterized actions, the policy can be represented as

$$\pi \text{Case}(s) = \max \left(\bigcup_{i=1 \dots m} B^{A_i}(\text{vCase}(s)) \right)$$

FOAPI iterates between two phases. In the first, the value function for the current policy is computed, i.e. the weights of the basis functions are computed using LP. The second phase computes the policy from the value function using the equation above. Convergence is reached if the policy remains stable between successive approximations. Loss bounds for the converged policy generalize directly from the ones for factored MDPs. The PRM approach by Guestrin too provides bounds on policy quality. Yet, these are PAC-bounds obtained under the assumption that the probability of domains falls off exponentially with their size. The FOALP bounds on policy quality apply equally to all domains. FOALP was used for *factored* FOMDPs by Sanner and Boutilier (2007) and applied to the SYSADMIN domain. Both FOALP and FOAPI have been entered the *probabilistic* part of the *international planning competition* (IPPC).

6.5.2.2 DYNAMIC REPRESENTATIONS

Approximate representations are useful, because they contain much fewer abstract states than exact value functions, and furthermore, they provide a stable structure on which parameter estimation can work. Still, the set of basis functions must be generated automatically or specified by a human expert, and will produce poor policies when chosen wrongly. An interesting approach is to generate the useful basis functions from the domain description itself, going from PIAGET-1 (or PIAGET-2) to PIAGET-3. Algorithms will be more complex, because now they have to generate structures too, but in return, the abstraction levels will be much better adapted to the problem at hand. Note that all exact algorithms for first-order IDP do both structure generation and parameter estimation.

One of the simplest types of approximations is to merge abstract states that have a *similar* value. For example, FODD-PLANNER uses ideas stemming from the SPUDD approach (Hoey *et al.*, 1999) to make abstractions smaller through approximations (Joshi and Khardon, 2008). A different approach is the approximation to the SDP approach described by Gretton and Thiébaux (2004a,b). The method uses the same basic setup as SDP, but the FODTR procedure is only partially computed. By employing multi-step *classical* regression from the goal states, a number of structures is computed that represent abstract states. The combination and maximization steps are not performed, but instead the structures generated by regression are used as a *hypothesis language* in the higher-order inductive tree-learner ALKEMY (Lloyd, 2003) to induce a tree representing the value function. We know from the exact methods that (especially in SDP) standard regression through deterministic action rules is computationally cheaper than a full FODTR step. A small set of examples driving the process of induction is generated using a planning algorithm in the ground problem. Because the generated features are generated through regression, and thus represent useful, reachable parts of the state space, not many examples are needed though. Not all structures generated through regression will be useful, but it is left up to the inductive step which ones are. The combination of deductive and inductive procedures has shown to perform very well on BLOCKS WORLD problems and logistics domains, and side-steps the difficulties encountered in the original SDP.

The same feature generation technique was later used in the *policy gradient* approach RPOG (Gretton, 2007a,b), and also in a slightly modified form in FOALP and FOAPI (Sanner and Boutilier, 2006, 2007). In the latter two systems, the idea is to use regression to extend the set of basis functions. If some portion of the state space φ has value $v > \tau$ in an existing approximate value function for some nontrivial threshold τ then this suggests that states that can reach this region (found by regression) should also have a reasonable value. However, since we have already assigned a value to $\$,$ we want the new basis function to focus on the area that is not covered by $\$.$ So, we negate $\$$ and conjoin it with $\text{Regr}(\$)$ yielding the new basis function $[\neg\$ \wedge \text{Regr}(\$) : 1; \$ \vee \neg\text{Regr}(\$) : 0]$. In FOAPI and FOALP there appears to be an exponential growth in running times when the number of basis functions increases. However, the fact that the feature generation algorithm produces orthogonal features (and the value function computations can exploit this), makes that the algorithm can handle more features.

A different approach to first-order basis function (or, *feature*) generation was proposed by Wu and Givan (2007a) (see also (Wu, 2007) and (Wu and Givan, 2007b)). A first difference with other feature generation methods is their form and semantics. Features in this method are first-order formulas with only one free variable, and only domain predicates are used in this representation. To evaluate a feature \mathbb{F} in a ground state $\$,$ the *number* of domain objects in $\$$ that satisfy \mathbb{F} are counted. For example, a BLOCKS WORLD feature might be $\exists Y_{\text{on}}(X, Y)$. The value of this feature is the number of blocks on top of other blocks, normalized by the total number of objects in the state. This entails that features are not binary-valued (as in other methods), but real-valued. The set of features is generated by a simple ILP algorithm, a *beam-search* in the feature space, guided by how well each feature correlates with the ideal Bellman residual. As in FOALP the features are used in an AVI algorithm, finding good weights for these features. An initial feature set can be arbitrarily chosen, and features can be added when needed. Learning progresses from simple to more complex domain instances, depending on measured progress on the instances.

The method is an upgrade of a propositional algorithm by the same authors (Wu and Givan, 2004, 2005), where propositional features are induced by a standard decision-tree learner. The method was tested on several IPPC problems, and in the game TETRIS.

Another form of approximation is introduced in the FOVI extension by Karabaev and Skvortsova (2005) (see also Hölldobler *et al.*, 2006; Skvortsova, 2006a, on its implementation FLUCAP) in which it is extended to *first order* LAO* (FOLAO*) with a heuristic search that avoids evaluating all states. FOLAO* is a direct first-order upgrade of the propositional LAO* algorithm (Feng and Hansen, 2002). Guided by an admissible heuristic, the search is restricted only to those states that are reachable from the initial state. First, it expands the best partial policy and evaluates the states on its fringe using an admissible heuristic function. Then, it performs DP on the states visited by the best partial policy, to update their values and possibly revise the current best policy (and iterate). The method is related to the REBP approach by Gardiol and Kaelbling (2003, see next section), by only considering the state space that is actually relevant for the given task. Furthermore, it is related to real-time DP (Barto *et al.*, 1995).

6.5.2.3 UPGRADING FROM SMALL INSTANCES

In all of the previous approaches, the working level of the algorithms is the abstraction level. Either parameters were estimated for logical structures, or the logical structures were constructed using a high-level model specification. In all these cases, abstraction and generalization were used *in* the process of generating solutions. An attractive alternative is to *separate* the process of generating a solution (e.g. a value function or a policy) from the generalization process. That is, one can first generate a solution for one or more (small) ground instances, and then use inductive generalization methods to obtain generalized solutions. This type of solution was pioneered by the work of Lecoeuche (2001) who used a solved instance of an RMDP to obtain a generalized policy in a dialogue system. A solved, ground RMDP can be used for supervised classification (e.g. when learning a policy) or numerical regression (e.g. when learning a value function). However, solving an RMDP can only be used for relatively small state spaces, and furthermore, for generalization to work, small RMDP instances must be 'similar' to larger instances. Note that the approach by Gretton and Thiébaux (2004a,b) that we have described in the previous section, is related to these upgrading methods too, though we have described it in the context of (first-order) feature generating, approximate IDP systems.

Yoon *et al.* (2002) introduced a method for inductive policy selection that learns an ensemble of decision lists that represents a policy. The algorithm can be viewed upon as an extension of the work by Martin and Geffner (2000) and Khardon (1999b) to stochastic domains. The policies obtained from solving several small domains are used to induce general policies that can be applied in larger problems. In fact, this algorithm has been used in the API framework (see Section 5.5.2) sharing the assumptions that policies often generalize well, if the policy language is well-chosen. Note that policy samples $\langle s, a \rangle$ in this framework are obtained from trajectories generated by a probabilistic planner.

Recently, García-Durán *et al.* (2008) too used a planner to generate policy examples, though only in deterministic settings. Generalization here employs a *relational nearest prototype classification* (RNPC) approach using the domain-independent, relational distance metric RIBL. Related to this, de la Rosa *et al.* (2008) devise the ROLLER algorithm that learns *generalized* policies from examples generated by a heuristic planner, using the re-

lational decision tree learner TILDE (see Chapter 4). Other related approaches that also reduce learning policies to *supervised learning* are (Lagoudakis and Parr, 2003; Benson, 1996; Khardon, 1999b,a; Martin and Geffner, 2000).

Two other methods that use complete, ground RMDP solutions are based on *value function* generalization (Mausam and Weld, 2003) and *policy* generalization (Cocora *et al.*, 2006; Kersting *et al.*, 2007). Both approaches first use a general MDP solver (SPUDD in this case, see Section 3.5), and both use a first-order decision tree algorithm to generalize solutions using relatively simple logical languages. One drawback of Mausam and Weld (2003)'s approach is that value functions do not generalize well over different-size domain instances. The policy induction approach by Cocora *et al.* (2006) however, delivers generalized policies that do generalize over multiple instances of an RMDP family (see Section 4.5.1.1). An interesting aspect of the latter approach is that it was applied in a realistic robot navigation domain that is represented in a relational language. Another advantage is that such a policy generalization approach might also be driven by policy samples that are obtained by other means. For example, one can let a human steer a (simulated) robot to generate (semi-optimal) sample state-action pairs (e.g. *behavioral cloning*, similar to Morales (2004a,b)'s approach), or use a probabilistic planning approach similar to Yoon *et al.* (2002)'s.

The *Relational Envelope-Based Planning* (REBP) (Gardiol and Kaelbling, 2003) too operates on the ground state space, but it focuses its attention to small, useful subsets of it. It lets an agent begin acting quickly within a restricted part of the full state space and to expand if resources permit. A set of probabilistic planning rules is used by a probabilistic planner to generate an initial sequence of states and actions, i.e. the envelope, leading to the goal. This sequence is turned into an RMDP, where obviously many actions lead outside the known state space. By equivalence-sampling from the states one step outside the envelope, the *fringe* states, the envelope is expanded. After that, the newly added actions are evaluated and a new policy (for the states in the envelope) is computed. The aim of REBP is to compute a policy, by first generating a good initial plan and use envelope-growing to improve the robustness of the plans incrementally. The approach was later extended by taking into account *action equivalences*, i.e. action instantiations that produce equivalent kinds of effects (Gardiol and Kaelbling, 2006b,a, 2007). A more recent extension of the method allows for the representation of the envelope using varying numbers of predicates, such that the representational complexity can be gradually increased during learning (Gardiol and Kaelbling, 2008).

6.5.3 Beyond the Markov Assumption

All methods presented so far assume full-observable first-order MDPs. Extensions towards more complex models beyond this Markov assumption are almost unexplored so far, and here we briefly mention some approaches that have went up this road.

Wingate *et al.* (2007) present the first steps towards relational knowledge representation in *predictive representations of states* (PRS, see Littman *et al.*, 2001, and Section 2.7). Although the representation is still essentially propositional, they do capture some of BLOCKS WORLD structure in a much richer framework than MDPs. Zhao and Doshi (2007) introduce a semi-Markov extension to the situation calculus approach in SDP in the HALLEY system for *web-based services*. Although no algorithm for solving the induced first-order SMDP is given, the approach clearly shows that the formalization can capture useful tem-

poral structures. Gretton (2007a,b) compute *temporally extended policies* for domains with non-Markovian rewards, using a *policy gradient* approach and we have discussed it in the previous chapter.

The first contribution to the solution of *first-order* POMDPs is given by Wang (2007). Although, modeling POMDPs using FOL formalisms has been done before (e.g. see Geffner and Bonet, 1998; Wang and Schmolze, 2005), Wang is the first to upgrade an existing POMDP solution algorithm to the first-order case. It takes the FODD formalism we have discussed in Section 6.5.1 and extends it to model *observations*, conditioned on states. Based on the clear connections between regression-based backups in IDP algorithms and value backups over belief states (see Section 6.1.5.4), Wang upgrades the *incremental pruning* algorithm (see Kaelbling *et al.*, 1998) to the first-order case, though it has not yet been accompanied by an implementation and experimental validation yet.

6.6. Discussion

Model-based methods have – in principle – access to all information necessary to find an optimal policy. On the contrary, the model-free methods in Chapter 5 lacked this information, and *sampling* formed the main computational bottleneck to compensate for this. However, one can not say that model-based are computationally less hard, because they do not need to sample. The main difference is that model-based methods rely heavily on *deductive* procedures rather than *inductive* ones. The model-based setting also comes along with an increased attention to *representation*, i.e. that of the model itself, but also of solution structures such as value functions and policies.

Our description of the IDP framework may be viewed upon as a *Rosetta stone* for DP with (structured) knowledge representation frameworks. IDP makes clear how DP algorithms can operate on the level of *sets* of states. Adding the use of knowledge representation formats to *describe* these sets, IDP provides us with a generic framework to do at least two things. One is to gain an understanding of what many algorithms in structured DP have in common. This provides opportunities for cross-fertilization between methods and to understand how particular knowledge representation languages influence how structure can be found in both representations and DP algorithms. A second thing IDP gives us, are ways to *upgrade* existing propositional algorithms to first-order domains. Because, once we see that first-order languages too describe sets of states, now being first-order interpretations in a particular FOL language, representations and operations can be found that replace their propositional counterparts. In the survey section of this chapter, we have seen many methods that are direct upgrades of propositional algorithms. Still, first-order IDP has to cope with new aspects, such as *families* of RMDPs. Our REBEL approach has been shown to be an IDP approach, using a relational state language with constraints. We have shown that we can compute compact, optimal value functions, even over infinite state spaces. Furthermore, we have described the use of tabling and policy induction mechanisms that make use of new opportunities in first-order domains.

abstract states	V_t									
	1	2	3	4	5	6	7	8	9	10
<code>bin(b, p).</code>	10.000	10.000	10.000	10.000	10.000	10.000	10.000	10.000	10.000	10.000
<code>tin(A, p), on(b, A), not_rain.</code>	8.100	8.829	8.895	8.901	8.901	8.901	8.901	8.901	8.901	8.901
<code>tin(A, p), on(b, A), rain.</code>	6.300	8.001	8.460	8.584	8.618	8.627	8.629	8.630	8.630	8.630
<code>tin(A, B), on(b, A), not_rain.</code>		7.290	7.946	8.005	8.010	8.011	8.011	8.011	8.011	8.011
<code>tin(A, B), on(b, A), rain.</code>		5.670	7.201	7.614	7.726	7.756	7.764	7.766	7.767	7.767
<code>tin(A, B), bin(b, B), not_rain.</code>			5.905	6.968	7.111	7.128	7.130	7.131	7.131	7.131
<code>tin(A, B), bin(b, B), rain.</code>			3.572	5.501	6.282	6.563	6.658	6.689	6.699	6.702
<code>tin(A, B), bin(b, C), not_rain.</code>				5.314	6.271	6.400	6.416	6.417	6.418	6.418
<code>tin(A, B), bin(b, C), rain.</code>				3.215	4.951	5.654	5.907	5.993	6.020	6.029
<code>tin(A, B).</code>	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Table 6.1: Logistics Domain Experiment: The t -th column shows the abstract state value function after the t -th iteration. When no value is given, the abstract state has not yet been created in that iteration. Bold numbers highlight changed values. See the text for more explanation.

Part III

Implications, Challenges and Conclusions

CHAPTER 7

Sapience, Models and Hierarchy

This chapter will argue that although existing methods – either model-based or model-free – are increasingly capable of solving ever more complex RMDPs, more generally intelligent systems will use such capabilities as separate skills and embed them in more sophisticated cognitive architectures that can do much more. For example, such architectures must incorporate several types of representation, reasoning and learning capabilities. In this chapter we will outline some of the characteristics of such architectures and study how existing RMDP solution techniques can contribute to such systems. We define the generic notion of a sapient agent, which has its foundations in logic and cognitive notions. Furthermore, we survey some approaches reported in the literature that can be used in such more general architectures.

SO FAR, WE HAVE BEEN FOCUSING ON SOLVING SINGLE PROBLEMS. Chapters 5 and 6 have extensively dealt with model-free and model-based techniques for solving specific RMDPs. Although we have presented the idea of *families* of RMDPs, which deals with a whole range of similar problems, the main goal of existing methods is to find a solution for a given RMDP. However, for generally intelligent systems it is highly unlikely that they will be employed for such a single task. Instead, RMDPs will have to be considered as – at most – subtasks in much more general tasks, requiring competence in areas ranging from perception, to reasoning, and to communicating to and learning from other intelligent entities. Learning how to solve a single BLOCKS WORLD objective, such as stacking all blocks into one tower, is useful to calibrate and evaluate a learning algorithm, but it is not expected that we would ever employ an intelligent system to do just that. For example, a generally intelligent robot that is employed in an office building has to cope with several tasks at once. It may have as its task to bring coffee to everyone, pick up the mail and distribute it, help people finding specific rooms in the building, and keep track of its battery level. Each of the individual tasks may be modeled as a single RMDP, but the robot has to combine, and presumably interleave, their solutions in order to be efficient. For example, it may have set as its goal to bring a certain coffee mug full of coffee to a certain person X. But at the same time, it might have to pause the execution (and learning) of the current task because its battery level has become dangerously low. Furthermore, it might reason about its other goals of bringing coffee to persons Y and Z and *schedule* them based on their position in the building, by using additional *background knowledge* about the building and connections between rooms, and

possibly some predictions about how much coffee will be left in the machine while it is executing its tasks and other people are taking coffee from the machine.

Dealing with multiple tasks simultaneously is one aspect of a more general intelligent system. In addition, it would have to cope with various forms of visual and auditory input, physical interaction with the environment and more, but here we restrict our discussion to the level of (logical) symbolic architectures. However, the dichotomy between systems that can perform a single, well-defined, task and systems that can do more general things, points towards the difference between *performance systems* and *general-purpose systems* in AI we have mentioned in Chapter 1. Nilsson (1995) summarizes it as follows:

”What I think is not needed (to give just one example) is a dynamic programming system for calculating paths of minimal costs between states in a Markov decision problem, yet some high-quality AI research is devoted to this and similar problems (which do arise in special settings). [...] The development of performance programs has focused AI research on systems that solve problems beyond what humans can ordinarily do. [...] What I am arguing for here is that these skills and knowledge bases be regarded as tools – separate from the intelligent programs that use them. It is time to begin to distinguish between general, intelligent programs and the special performance systems, that is, tools that they use.”

Intelligent agents have already found their way to the public, and during the last decades many intelligent agents have been shown to be effective in solving particular tasks (Wooldridge and Jennings, 1995). However, an intelligent agent is often used for a single task only, e.g., think about a CHESS playing program which is only used to play CHESS. If an agent has to fulfil multiple tasks, more complicated issues arise, such as a decision method for choosing the current goals based on current information about the environment and refining the decision method based on a learning capability. Making the transition from intelligent agents to agents that decide autonomously and learn to refine their decision making capability, requires new types of agents.

In this chapter we aim at characterizing how the work on RMDPs can be *upgraded* to be used as a *tool* (in Nilsson’s words) in more generally intelligent systems. Many such *cognitive architectures* exist (e.g. see Langley *et al.*, 2006; Langley, 2006; Vernon *et al.*, 2007, for recent overviews) , but here we focus on a more *generic* approach, rooted in cognitive, logic-based agent architectures over mentalistic concepts such as beliefs, goals, and plans, termed *sapient agents* (van Otterlo *et al.*, 2003, 2007). Sapient agents represent a form of middle ground between representationally simple formalisms aimed at solving individual RMDPs on the one hand, and general-purpose AI architectures on the other.

Goals and Outline of this Chapter. In contrast to the previous chapters, the current chapter is less concerned with the development of formal theories and algorithms. Instead, this chapter is about the characterization of the general contours of scaling up to problems that lie beyond single RMDPs. We discuss two topics in this chapter. The first is about describing how the capability of solving single RMDPs fits into more general *cognitive architectures*. We discuss the concept of *sapient agents*, and pinpoint several issues that are important for a generally intelligent agent. The second topic is about surveying what has been reported in the literature about several specific capabilities of such sapient agents, such as the *transfer of knowledge* and *hierarchical behavior decompositions*.

7.1. Scaling Up

Scaling up beyond the level of individual RMDPs requires a more complex architecture than the general FORM definition of Chapter 4. To address more general issues in decision making, we need the notion of a *cognitive architecture*. We will look upon sapient agents, from the starting perspective of cognitive agents extended with (relational) RL capabilities. A cognitive agent is assumed to have some internal state consisting of mental attitudes such beliefs, goals, and plans, receives inputs through its sensors, and performs actions. The actions are decided on based on its mental state in such a way that its effort to attain a goal will be minimal. We stay as close as possible to the *logical* context with which we have been concerned throughout most of this book. Cognitive architectures based on logic offer many opportunities to see all the agent's reasoning as pure logical deduction. As Reiter (2001, preface) puts it strongly: *[W]hen faced with a dynamical system that you want to simulate, control, analyze, or otherwise investigate, first axiomatize it in a suitable logic. Through logical entailment, all else will follow, including system control, simulation and analysis.* However, many other types of reasoning mechanisms and representational devices could be useful in addition to logical deduction, such as neural networks, fuzzy logic, probabilistic reasoning, Bayesian networks, etc. We consider an agent in which all of these mechanisms may run in parallel. For example, perception (e.g. pattern recognition) may be done using neural networks, whereas communication could best be done using logical languages. Here though, we focus mainly on the *internal* reasoning mechanisms of a cognitive agent, implemented using generic, logical formalisms.

One important aspect of cognitive architectures is that they enrich the *mental state* of the agent far beyond just its current observations (e.g. an RMDP state). A second aspect is that they introduce various interactions between *declarative* and *procedural* knowledge. A third aspect of architectures that are more general, is the dichotomy between *learning* and *reasoning*. We will discuss these issues briefly in the following.

7.1.1 Extending Mental States

The large majority of the approaches we have described in the previous chapters uses a very simple notion of *state*. For Markov systems such as MDPs and RMDPs, the state is the (fully-observable) current state of the environment. For POMDPs, the notion of a state is slightly more complex, and usually consists of a probability distribution over environment states. An agent's mental state consists further of a number of simple structures that represent value functions, transition functions, reward functions and policies. Most often, the mechanisms that drive the agent's behavior are simple too; it observes the current state, computes an action and performs it, receives a reward, observes the resulting state and updates some of its representations (e.g. of a value function). Some approaches do use some more complicated mechanisms to generate and modify parts of the mental state (e.g. the structure of the policy) as part of the behavior. A more generic decomposition of an agent is based on commonly used components of general *cognitive architectures*, and consists of at least four main parts (see Langley, 2006).

1. **Memories.** Different types of memory, both short-term and long-term, must be present to store the agent's beliefs, knowledge, and goals.
2. **Organizational structure.** Structures that are present in the memory are repre-

sented in a specific format (e.g. logical rules, number tables, etc.) and their organization is characteristic for a particular architecture.

3. **Functional Processes.** The way structures in the memory are used for computations, modified, deleted or added, is determined by a functional process that iterates through the organizational structure of the memory, using those elements that are required for specific tasks. Note that both the behavior and additional learning mechanisms are part of this process.
4. **Programming Language.** To build individual agents for specific tasks, a programming language is required to construct knowledge-based systems that adhere to the architecture's structure and constraints.

As we can see, much of what we have focused on until now, was based on simple memories that store e.g. value functions and policies, and the functional process amounts to the basic iterative RL methodology that modifies these structures based on interaction with an environment, mostly in a reactive fashion. However, a crucial step has been the employment of FOL languages, which introduce excellent opportunities to scale up to these more general cognitive architectures that are often based on such languages.

7.1.2 Declarative versus Procedural Representations

A cognitive architecture can employ many types of representational devices, ranging from propositional to first-order, and from rules to neural networks. In such architectures, an important general distinction is that between *declarative* and *procedural* representations. Declarative encodings of knowledge can be manipulated by cognitive mechanisms independent of their content. First-order logic is a classical example of such a representation. For example, throughout the previous three chapters we have often used declarative background knowledge about the BLOCKS WORLD, for example to specify a clause as `onTop/2`. Such knowledge can be used as input for a deduction algorithm that employs it to derive properties of states, but it can also be employed as input for ILP algorithms to induce new structures from data. Generally speaking, declarative representations support very flexible use, but they may lead to inefficient processing (see Chapter 4 on logical reasoning mechanisms). In contrast, procedural formalisms encode knowledge about *how* to accomplish some task. *Production rules* are a common means of representing such procedural knowledge. In general, procedural representations let an agent apply knowledge efficiently, but typically in an inflexible manner, and more specific for a particular task.

Note, however, that both definitions are not mutually exclusive. The difference between both depends less on the exact format, and more on the type of architectural mechanisms that can access it. For example, production rules can be viewed as declarative if other production rules have access to them and can inspect them (structurally). Another way of looking at this distinction is that declarative knowledge is generally more *stable*, and moreover, is not built into the system. Because of this, declarative knowledge is much easier to *transfer* to other tasks, or to communicate to other, intelligent systems¹.

¹This is similar to the fact that it is much easier to communicate *facts* (i.e. *what* is the case) than procedures for doing something (i.e. *how* something comes about), or explanations. The latter are much more dependent on the details of a specific architecture.

A related distinction between different types of knowledge in a cognitive architecture is that between *skill* knowledge and *conceptual knowledge*. The first is related to the procedural knowledge and focuses on how to plan sequences of (mental or physical) actions to perform certain tasks. Conceptual knowledge is more related to *perception* and the construction of *concepts*, *categories*, and *classes* of objects, and is a less action-oriented type of knowledge. Because of the context of this book, in which we mainly deal with pre-defined symbols and, in fact, action-oriented tasks, we do not deal with this type of knowledge, nor how it comes about.

7.1.3 Learning vs. Reasoning

So far, we have been mainly concerned with *reactive* tasks, in which we have focused on the learning mechanisms, without paying much attention to *reasoning*. Some (logical) reasoning is always part of relational RL, for example to find a matching abstract state for the current environment's state, to employ background knowledge predicates, or to apply a STRIPS rule to the current state, but in general the amount of reasoning has been relatively limited. In contrast, the model-based setting (see Chapter 6) uses extensive reasoning, though no actual experience-based learning is involved. Dietterich (2003) discusses reasoning in the general ML setting and explains why reasoning is often neglected for a large part in any of the supervised, unsupervised, or reinforcement learning settings.

"Most work in ML has centered on learning policies rather than declarative knowledge. Hence, as the critics point out, ML has avoided the need for an inference engine (and it has avoided the computational cost of reasoning). Even in cases where ML has studied learning declarative knowledge, it has learned knowledge only for very simple inference engines. There are many reasons for this. First, the ML community has focused on end-to-end performance, and this naturally leads to an emphasis on learning policies whose end-to-end performance can be directly evaluated (i.e. without dealing with an inference engine). Second, there is a statistical component to ML. Consequently, the inference engine must be probabilistic, and probabilistic inference is more costly and more complex than logical inference. Third, ML has studied only the simplest form of experience for learning: pairs of input observations and output actions. It has (for the most part) not exploited richer inputs such as natural language queries, explanations, instructions, and so on. There are severe statistical limitations to what can be learned from input-output pairs alone." (Dietterich, 2003)

This chapter deals with the first argument Dietterich mentions. By looking at more general architectures that can handle more complex tasks, and thus moving away from performance on an isolated task, reasoning becomes more important quite naturally. The second argument is related to the recent progress in SRL (see Section 4.3.3). Nowadays more and more efficient probabilistic reasoning engines for first-order settings become available, opening up possibilities for additional reasoning in these contexts. On the other hand, the increased popularity of statistical and probabilistic methods has reduced the fragility of traditional symbolic schemes, but only at the cost of a great loss in representational power. Some subfields in ML, but also in natural language processing, have mostly given up on the goal of interpretable symbolic representations, and care mainly about performance. We have seen similar approaches in relational RL, such as the use of kernels

for value function approximation in Chapter 5. The third argument of Dietterich still applies to most systems, though there are some examples of ML systems using more complex inputs, such as *learning from proof trees* in the ILP setting (see De Raedt and Kersting, 2004). However, even though we focus on more general architectures in this chapter, we still assume that the agent learns from pairs of observations and output actions mainly.

This book is concerned with opening up the cognitive structure of the agent in RL, and identifying various sorts of representations and reasoning mechanisms. We have seen a progression starting from atomic states (Chapter 2), to propositional states (Chapter 3), to first-order representations (Chapters 4 to 6) and finally towards cognitive architectures. Interestingly, this resembles similar (and opposite) shifts in psychology, where the *behaviorist* movement emerged as a reaction to *cognitive* psychology, rejecting the postulation of such internal, cognitive structures. In AI we have seen such shifts for example when *behavior-based* approaches (Brooks, 1991; Arkin, 1998) emerged that reacted against symbolic approaches in AI.

7.1.4 Examples of Existing Formalisms

There are many systems and formalisms that incorporate many of the notions mentioned above. The agent literature (Wooldridge, 2002; Weiss, 1999; Ferber, 1999) contains many examples of logical agents capable of reasoning, communicating, planning and acting. As cognitive architectures typically come with an associated programming language for use in building knowledge-based systems, there is much overlap with the research on agent programming languages. Relational RL methods can provide a general framework for *learning* in agent architectures (see also Džeroski, 2002; van Otterlo *et al.*, 2003; Tuyls *et al.*, 2005, for pointers).

Decision-theoretic (agent) programming languages are powerful, first-order languages that can be used to axiomatize dynamic worlds. One of the most commonly used languages is *situation calculus* (Reiter, 2001; McCarthy, 1963). Its stochastic version (Reiter, 2001; Boutilier *et al.*, 2000b) has been used in several model-based approaches for RMDPs, including *symbolic dynamic programming* (Boutilier *et al.*, 2001). The GOLOG-Language (Levesque *et al.*, 1997) based on situation calculus, can be used to specify *complex actions*, consisting of standard programming language constructs such as *conditional actions* and *procedures*. In fact, this can be seen as providing a policy space bias, or *program constraints*. Gu (2003) highlights the relation between macro-actions in situation calculus and hierarchical abstraction of (R)MDPs. Finzi and Lukasiewicz (2004a) extend GOLOG with *game-theoretic* constructs to deal with *multi-agent* domains. A related agent programming language approach is the FLUX system (Thielscher, 2005) based on the *fluent calculus* (Thielscher, 1998). We have seen many approaches in Chapter 6 that use these languages to solve MDPs specified over first-order domains. Making use of their existing extensions would enable to solve more complex and structured problems. Other decision-theoretic agent programming languages that can be used to specify MDPs over first-order domains are, for example, the *integrated Bayesian Agent Language* (IBAL) (Pfeffer, 2001), the *independent choice logic* (ICL) (Poole, 1996) and ALISP (Andre and Russell, 2002) but there are many more.

GOLOG and FLUX are examples of formalisms that are based on particular, logical systems, and their syntax is closely linked to the framework's representational assumptions. Many more cognitive architectures exist that are based on various other types of

formalisms, such as ACT, PRODIGY, CLARION, EMILE and many more. Two so far, have approached problems similar to relational RL. ICARUS (Choi *et al.*, 2004) is a cognitive architecture in which – among many other things – hierarchical, relational skills can be specified and learned. These skills support reactive execution and model-free RL can be used over the skill hierarchy. However, recent work has extended this with model-based learning techniques (Langley *et al.*, 2004). Earlier, (Shapiro and Langley, 2002) described SHARSA as an extension of SARSA, adding (H)ierarchical aspects. In ICARUS, a clear separation exists between control knowledge in terms of logical skills and numeric utility functions. A driving simulation shows that this enables learning a variety of agent behaviors (e.g. different driving styles) based on the same set of skills. Yet another powerful cognitive architecture is SOAR. SOAR is strong at knowledge-rich, symbolic reasoning but weak at knowledge-lean, statistical-based learning. Recently, Nason and Laird (2004a,b) incorporated elements of relational RL into the SOAR architecture, creating SOAR-RL. SOAR-RL employs model-free RL (SARSA) in order to learn preferences for operators. Wang and Laird (2007) investigated the effects of putting an action history in the state representation in the same approach.

7.2. Characterizing Sapient Agents

The term *sapient agents* denotes a general kind of agent that can handle different tasks simultaneously, and is presumably situated in an environment, involved in a sustained activity over longer periods of time. Sapient agents are assumed to have accumulated learning and knowledge, the ability to discern inner qualities and relationships, often called the agent's *insight*, and good sense or *judgments*. These concepts and properties remain, however, intuitive and informal without explicit formal semantics. In this section, we consider sapient agents as a specific type of *cognitive agents* for which many formalizations have been proposed. In particular, we believe that properties such as knowledge, insight, and judgments of sapient agents are related to, and should be defined in terms of, *mentalist* concepts such as *beliefs*, *goals*, and *plans* as used for cognitive agents. Therefore, we propose an interpretation of properties of sapient agents based on mentalistic concepts of cognitive agents and identify certain problems such as the integration of learning and decision making processes that together influence the behavior of sapient agents.

We assume that insight and judgment properties of sapient agents determine their course of actions. For cognitive agents the course of actions can be specified in terms of their mental attitudes which contain at least beliefs, goals, and norms, capabilities such as actions and plans, reasoning rules that can be used to reason about the mental attitudes, communication, and sensing. Given the above mentioned entities, the decision making ability of agents can be considered as consisting of *reasoning* about mentalistic attitudes, selecting goals, planning goals, selecting and executing plans, among other things (Dastani *et al.*, 2003a).

In our view, the judgment of an agent can be considered at the lowest level as making choices about how to reason about its mental attitudes at each moment in time. For example, an agent's judgment can be established by reasoning about its goals or by reasoning about its goals only when they are not reachable using any possible plan. Some more moderate alternatives are also possible. For example, the agent can create a plan for a goal and execute the plan. If this leads to a stage where the plan cannot be executed any further,

then the agent can start reasoning about the plan and revise the plan if necessary. If the goal can still not be reached, then the agent can revise the goal. So, this leads to a strategy where one plan is tried completely and if it fails the goal is revised or even abandoned. In general, an agent with the judgment ability should be able to control the relation between plans and goals. For example, an agent should control whether a goal still exists during the execution of the plan to reach that goal. If the corresponding goal of a plan is reached (or dropped), the agent can allow or avoid continuing with the plan.

We consider the insight of agents to be directly related to the ability of agents to evaluate their mental states and mental capabilities. We therefore assume that the insight ability is the ability to *learn* how to reason about mental attitudes and thus how to make decisions. The reasoning capability determines the agent's decision making behavior and the learning capability determines the evolution of the reasoning capability through experiences. The focus of this section with respect to the agent's insight is on the aspects of the agent's mental state and mental capabilities that are influenced by the learning process. These could be the goals, beliefs, desires, reasoning rules of even basic capabilities (the agent can learn new actions).

7.2.1 Cognitive Agents

In this section, we consider some aspects of the mental states and mental abilities of cognitive agents that may evolve through learning and from experiences resulting in properties associated with the sapient agents. In general, cognitive agents are assumed to have mental states consisting of mental attitudes such as beliefs, goals, plans, and reasoning rules (Rao and Georgeff, 1995, 1991; Rao, 1996; d'Inverno *et al.*, 1997; Hindriks *et al.*, 1999; Broersen *et al.*, 2002). For example, a cognitive agent may *believe* there is no coffee available, *desire* to drink coffee, and *desire* to have tea *if* there is no coffee. Moreover, the behavior of cognitive agents, i.e. the actions it chooses and performs, are assumed to be determined by deliberating on the mental attitudes (Rao and Georgeff, 1995; Dastani *et al.*, 2003a,b). The deliberation process is a *continuous* and *iterative* process that involves many choices and decisions through which actions are selected and performed. For example, a deliberation process may select one goal, plan the goal and execute the plan. If the goal cannot be planned, it may either drop the goal or revise it. The revised goal may be planned. It is also possible that a plan cannot be executed since some of its constituting actions are blocked. In such a case, the agent may either decide to drop the plan or revise it. The existing proposals of cognitive agents (Rao, 1996; d'Inverno *et al.*, 1997; Dastani *et al.*, 2003a; Hindriks *et al.*, 1999; Broersen *et al.*, 2002) assume that many of these choices and decisions are fixed. These choices and decisions are based on predefined criteria and remain unchanged during agent's lifetime. In this section, we introduce a general architecture for cognitive agents and discuss possible choices and decisions that are involved in the deliberation process.

7.2.1.1 A COGNITIVE AGENT ARCHITECTURE

We consider a general architecture for cognitive agent consisting of the representation of mental attitudes and the deliberation process. Such an agent architecture is depicted in Figure 7.1. According to this architecture, an agent *observes* the environment and *communicates* with other agents. The observation of an agent provides the facts that the agent

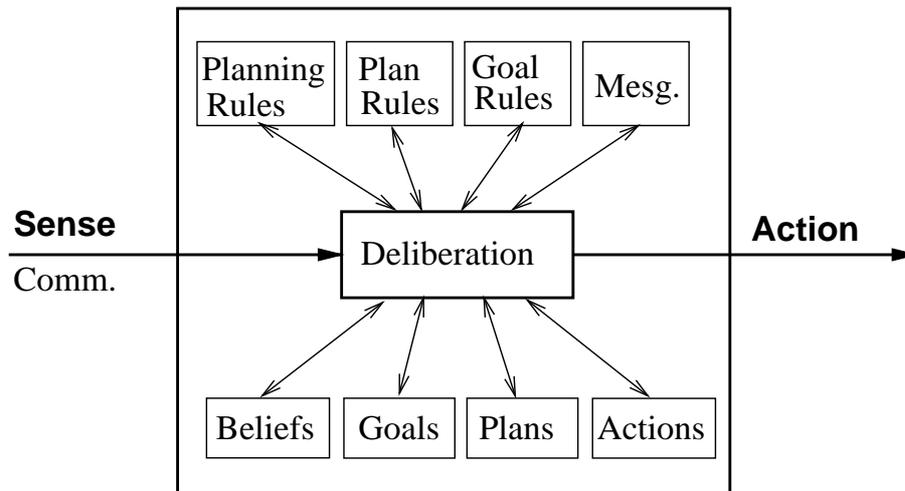


Figure 7.1: A general architecture for cognitive agents.

recognizes from its sensory information. These facts can be used to update the agent's *mental state*. The communication provides information that an agent receives from other agents. The received information are messages that are stored in the message box (Mesg.) of the agent. These messages can be represented in terms of the identifier of the sender and receiver, a logical sentence that determines the content of the message, and a performative which indicates the modality of the message, i.e. whether the content is meant to inform the receiver, contains requests for the receiver, and so on.

The *beliefs* of an agent represent its general world knowledge as well as its knowledge about the surrounding environment. The beliefs are usually represented by sentences in a FOL language. The *goals* represent the states that the agent *desires* to reach. Like beliefs, goals are represented by sentences of a logical language as well. *Actions* represent basic capabilities that an agent can perform. These actions can be cognitive actions such as belief updates, or external (physical) actions such as communication or movement actions. The actions are usually specified by pre- and post-conditions which are belief formulae. The plans represent structured patterns of actions that an agent can perform together.

7.2.1.2 ACTING, PLANNING AND DELIBERATING

A planning rule expresses that a goal can be achieved by performing a plan under a certain belief condition. A planning rule has the form $\phi \leftarrow \beta \mid \pi$, which indicates that goal ϕ can be achieved by plan π if belief condition β holds. A goal rule determines how to modify a goal under a certain belief condition. A goal rule has the form $\phi \leftarrow \beta \mid \psi$ which indicates that goal ϕ can be revised as goal ψ if belief condition β holds. Likewise, a plan rule determines how to modify a plan under a certain belief condition. A plan rule has the form $\pi \leftarrow \beta \mid \pi'$ which indicates that plan π can be revised as plan π' if belief condition β holds.

Cognitive agents deliberate on these concepts to decide which actions to perform at each moment of time (Dastani *et al.*, 2003c). The deliberation process involves many activities such as applying a reasoning rule for the above mentioned purposes, selecting a goal to achieve, selecting a plan to execute, generating a plan to achieve a goal, etc. In particular, a cognitive agent decides at each moment of time which activity to perform. It should be noted that different applications require different deliberation processes and that there is not one single universal deliberation process. An example of a deliberation

process is the iterative procedure in Algorithm 24. Note how much more complex such a

Algorithm 24 A generic deliberation cycle of a cognitive agent.

- 1: **repeat**
 - 2: find and apply goal rules that are applicable
 - 3: find and apply plan rules that are applicable
 - 4: find a goal and a planning rule which is applicable to it
 - 5: apply the selected planning rule to the selected goal
 - 6: find and execute a plan
 - 7: **until** a sufficient stopping criterion is satisfied.
-

deliberation cycle is when compared to a standard, online RL algorithm (see Algorithm 3), both in terms of representational aspects, as well as reasoning mechanisms. In order to specify, design, and implement a cognitive agent one needs to initialize its cognitive state and specify, design, and implement various decisions and choices involved in the deliberation process beforehand. For example, the agent designer should develop beforehand various selection functions to select goals, plans, and various types of rules at various stages of the deliberation process. Also, the agent designer should indicate beforehand how goals and plans are generated and dropped. For many types of agents, especially sapient ones, it is not possible, or even desirable, to specify all these concepts at design-time. Therefore, sapient agents should be capable of learning.

Emotions. Emotions will also be important for truly sapient agents. Emotional attitudes towards agents, objects, events etc. can become important in the process of acting, planning and deliberation. Emotions motivate and bias behavior, but they do not completely determine it. They play a *reflective* role in decision making and learning (Picard, 1997), may monitor planning and may be *prospect-based* (Ortony *et al.*, 1988). By focusing on *emotion-inducing* events, the agent can decide more effectively. Basic emotions such as fear can trigger behavior needed to act fast, or to quickly change plans. Emotions such as happiness can influence choices for certain goals or plans. In some sense, emotions complement *ratio* so that the agent becomes wiser, more sapient. It is acknowledged that, at least in humans, emotions are not a separate process from cognition, but both are inextricably intertwined. It can even be stated that without emotions, decision-making and acting is hardly possible and that reason itself uses emotions to guide its decision making processes (Damasio, 1994). Even though some may argue that it is not important for machines (agents) to actually *have* emotions, it surely is important to be able to *reason about* emotions, especially in situations in which natural language understanding and cooperative problem solving are important. When interaction with humans is involved, a capability to deal with emotions, whether to express or to understand, becomes highly desirable or even needed (Picard, 1997).

Many systems in AI today deal with emotions, though in very different ways. For cognitive agents it would be best if emotions are based on the cognitive architecture and logic, and more importantly, be about its beliefs, intentions and goals. An example of such a system was proposed by Dastani and Meyer (2006), who define four types of emotions (happiness, sadness, anger, fear) in terms the architectural aspects of a cognitive agent, and who incorporate these emotions into the deliberation cycle. Let us, without going into detail on the syntax and semantics of the system, give an example of the notion of

happiness, defined in terms of the agent's intentions, goals, and beliefs:

$$\mathbf{I}(\pi, k) \wedge \mathbf{C}(\pi) \wedge \alpha \preceq \pi \wedge \mathbf{B}([\alpha]k' \rightarrow [\alpha](\mathbf{B}k' \wedge \mathbf{C}(\pi \setminus \alpha)) \rightarrow \mathbf{happy}(\pi \setminus \alpha, k, k'))$$

More informally, it states that an agent that is *happy* observes that its subgoals are being achieved. In particular, an agent that has the intention to do π for achieving goal k (denoted $\mathbf{I}(\pi, k)$), and is committed to it (\mathbf{C}), and that believes (\mathbf{B}) that by performing the initial part α the subgoal k' should be achieved, is happy (with respect to the remainder $\pi \setminus \alpha$ of the plan - to which it is still committed, the goal k and subgoal k') if after the performance of α it believes that indeed the subgoal k' has been achieved. Such formalizations make emotions just another part of the reasoning process, and it indicates that emotions can be used to program behaviors in different ways.

7.2.2 Learning in Cognitive Agents

There is general agreement nowadays that intelligent agents should be *adaptive*, i.e. capable of learning. For learning to work, agents should be able to make the proper generalizations to reuse learned knowledge to apply it to new situations similar to encountered ones. For sapient agents learning as a capability should be extended to learning how to organize the deliberation cycle. Sapient agents can learn how to solve multiple tasks in parallel, how to deal with multiple goals and also how to set the right priorities. They can use their own experiences but also the social context for doing this. The actual acquisition of new knowledge can be performed both *analytically*, for example by reflecting on its current beliefs, knowledge and its deliberation cycle, but it can also be based on empirical data, gathered while interacting with the environment or other agents. RL is a most prominent candidate for learning action-oriented knowledge, though other types of unsupervised and supervised learning methods may be employed for categorization and prediction purposes.

In Section 4.4 we mentioned the fact that in cognitive agent architectures, many aspects are specified beforehand. However, for a sapient agent we believe that various choices and decisions involved in the deliberation process should be learned through experience instead of being fixed and defined beforehand.

7.2.2.1 LEARNING AND ADAPTING THE DELIBERATION CYCLE

RL is a general methodology to learn optimal policies by interacting in an environment. For sapient agents, we can use any kind of algorithm from Chapters 5 and Chapter 6 to optimize single behaviors. The use of FOL languages supports a direct application of relational RL in the cognitive structure of the sapient agent. In addition to learning to select actions, as in individual RMDPs, the sapient agent should learn other types of structures in its cognitive architecture. In particular, the agent should learn at run time various concepts ($\text{car}(X)$), facts ($\text{car}(p)$), and rules (e.g. $\text{bird}(X) \rightarrow \text{fly}(X)$) that constitute its beliefs. Moreover, given goal formulae ϕ and ψ , plan expressions π and π' , and belief formula β , the following can be the subject of learning with regards to the agent's goals and plans:

- Which goal to select in order to plan, and which plan to select in order to execute? Two types of selection functions can be learned. The goal selection functions should be learned based on the agent's beliefs and the plan selection functions should be learned based on the agent's beliefs and goals.

- Which goal or plan to generate in certain situations? This can be achieved by learning goal or plan rules of the form $\top \leftarrow \beta \mid \phi$ and $\top \leftarrow \beta \mid \pi$, respectively.
- Which goal or plan to drop in certain situations? This can be achieved by learning goal or plan rules of the form $\phi \leftarrow \beta \mid \top$ and $\pi \leftarrow \beta \mid \epsilon$, (where ϵ is the empty plan) respectively.
- Which goal or plan to modify in certain situations? This can be achieved by learning goal or plan rules of the form $\phi \leftarrow \beta \mid \psi$ and $\pi \leftarrow \beta \mid \pi'$, respectively.
- How to plan a goal in a certain situation? This can be achieved by learning planning rules of the form $\phi \leftarrow \beta \mid \pi$.

Finally, for each type of rules a selection function should be learned that selects a rule to apply at each moment of time. These selection functions should be learned based on agent's mental state and differ for each type of rules. In particular, the selection function for planning rules should be learned based on agent's beliefs, goals, and plans, the selection function for plan rules should be learned based on agent's beliefs and goals, and the selection function for goal rules should be learned based on agent's beliefs. In the following section we discuss how these aspects can be learned by various learning techniques.

7.2.2.2 LEARNING GOALS, PLANS AND CONCEPTS

In order to cope with the demands of a sapient agent described in the previous section, we can use hierarchical RL (using relational representations). We consider goal-selection and plan-selection first.

For goal-selection, the agent has to map its beliefs to a particular goal which it will adopt. Each time-step the agent can change its mind about the goal, but in order to allow the agent to continue with one particular goal, we can use a mapping from beliefs and the previous goal to a newly selected goal. RL algorithms learn value functions for this by trial and error using e.g. Q -learning (Watkins and Dayan, 1992). The goal is to learn to select goals leading to the maximal average reward intake per time-step. By trying out goals, and using plans or actions to achieve these goals, the agent gets estimates about the quality-value (Q -value) of selecting each of its goals given some mental states. Since the agent can at any time change its goal, it can drop previous goals and continue with new ones. It can also learn that committing to some goal is good until some mental state tells the agent to adopt another goal. Thus, using the hierarchical RL framework, selecting and revising goals may be learned just as learning action sequences.

For learning to select plans, the agent has to map a goal and beliefs to a plan. There can be multiple plans, and some plans may even consist of single actions. Although some plans take longer than single actions, this is not any problem if hierarchical RL is being used. The agent can even choose to invoke a planner which will then plan at a specific time-step. If this planner returns useful plans given some mental state, it will be invoked more often in that context. Plans can be dropped or revised at any time, since the agent selects a plan or action at each time step. Thus, again using the hierarchical RL framework, selecting and revising plans can be learned just as learning action sequences.

Finally, the agent has to learn to map sensory information obtained by for example cameras to concepts. This can be done by using pattern recognition methods such as neural

networks or support vector machines. Each time the agent receives sensory information and does not understand what it sees, it should get feedback about the concept it is looking at. This can only be done in a social setting in which humans communicate with the agent, and the agent is also able to communicate with other agents. We will examine this issue further in section 7.2.3.

Emotions may influence behavior as was explained previously. Emotions may also influence learning. For example, a negative emotion that produces a bad feeling may trigger reassessment of what causes the bad feeling, followed by learning how to avoid it in the future. A sapient agent can also predict that by not doing an action, it will feel even worse, and by feeling this, it can interrupt its current behavior to do that action. Emotions can also help focus on a goal, or trigger to reassess a situation (e.g. by *insight*, *reflection*) and look for a way to improve it, thus adapting the behavior.

7.2.3 The Social Environment

Agents, especially sapient ones, will usually be *situated* in complex, multi-agent, social environments in which they have to interact with other agents and humans. Such complex environments create difficulties, but also opportunities, especially in learning. We will discuss some of these in this section.

RL has already been applied successfully for solving particular multi-agent problems such as network routing (Littman and Boyan, 1993), elevator control (Crites and Barto, 1996), and traffic light control (Wiering, 2000). For all these problems, the agent still has to solve a particular task such as controlling a specific traffic light and therefore these agents are not sapient at all. We can use multi-agent systems to make it easier for agents to learn to become sapient agents, however.

If the agent has to learn to achieve a goal and it can choose which task to learn, there are several complicated issues. In some sense, the agent has to devote its time to learn something useful. But what if the agent is unable to learn to solve a particular task? When should it stop trying to learn the task? And also, how much reward can it expect when it would be able to learn to perform the task? A solution is to let the agent learn from other agents. For example, the agent can estimate its learning time by looking at other agents, or by communicating with them. The agent can also ask the reward functions of other agents, it can estimate the learning time by asking or looking at the other agent, and the agent can even ask the decision skill to solve a particular task to another agent. Thus, some issues seem complicated, but may become easier when the agent is not alone in the world. Although the whole system would become much more complex, particular subproblems are easier to solve. Some problems would even be impossible if the agent cannot learn by imitating other agents. For example, suppose one agent, a robot, approaches a deep canyon and just near the edge, it slips and falls into the canyon. Because of the fall, the agent is destroyed and it discontinues to exist. The only way of learning that one should not come too close to the edge of the canyon is to look at the results of other agents approaching it. Since it is easy to see that coming too close to the edge of the canyon was bad for the other agent, a sapient agent can learn that this is a wrong action in this context.

In multi-agent settings we have to distinguish between competitive, co-operative, and semi-competitive settings. Although we would like all agents to be cooperative, this is not realistic since each agent tries to maximize its own average reward intake per time-step.

However, even in (semi-)competitive settings it makes sense to let agents communicate (e.g. if two agents try to walk through the same corridor and bump against each other, they can signal to which side they will go). By communicating knowledge, agents can share experiences, concepts and procedural knowledge of how to solve tasks. Using communication, possibly with humans, is also a good way to get a lot of examples for learning to classify sensory information into concepts. These are examples of *learning by communicating*. Classical experiments show that in many cases, communication between agents can have a positive influence on learning behavior, provided that communicated information is useful and not superfluous (Tan, 1993). On the other hand, agents can also *learn how to communicate* (Weiss, 1999). This involves learning *what, when, with whom* and *how* to communicate. Social laws, protocols and shared *ontologies* are important factors in communication.

Agents can learn to judge just like other agents, and agents can reward each other using ethical or social laws which have already existed for a long time and therefore may be evolved or preprogrammed. Thus, in multi-agent systems judgment and insight can also be learned, obtained, and refined using communication. For communication between agents some issues such as trust (insight in relationships) play an important role and have to be learned based on the experiences of the agent. If another agent provides wrong estimates about the learning time or reward for solving a particular task, or it gives a wrong decision skill for solving the task, the agent can learn that this agent cannot be trusted. The agent can also ask other trusted agents, whether they trust another agent. In this way social relationships among agents can evolve.

The problem of using reward functions is that it is difficult to say how much reward one should get for task *A* relative to task *B*. The decision of the agent will be to do the task leading to maximal average reward per time-step. However, if these relative reward values are incorrect, the agent could always do one single task at which it is good. Therefore the reward function should also be dynamic, where a reward is given only under particular circumstances. The reward could be made dependent on the agent's emotions such as boredom, pride, pity, disappointment, satisfaction, or anger. In this way, an agent who is angry with another agent may learn not to communicate interesting information. Also if the agent is bored with its current task, it will get less reward for doing it, and therefore may switch to another goal.

7.2.4 Discussion of the Sapient Model of Agents

In this section we have given a characterization of *sapient agents*. By starting from the notion of a *cognitive* agent, for which many formalizations exist, we place cognitive notions such as *beliefs, desires, goals, and plans* at the core of the deliberation cycle of a sapient agent. Furthermore, with this as a starting point, we have a firm basis for a model of true *sapience* as well as that we can take advantage from existing knowledge and formalizations concerning the modeling of cognitive notions, logic-based systems and agent programming languages such as 3APL (Hindriks *et al.*, 1999). Furthermore, we have emphasized the need for managing control over different tasks that can be performed in parallel, by choosing constantly between actions, goals and plans in the deliberation cycle. Various tasks can also be run in parallel on different cognitive levels. On the perceptual level, pattern recognition can transform visual images to (logical) concepts, while planning and acting can be performed on a higher cognitive level. We have also stressed the importance of

emotions as a possible factor in both behaving and learning. In a single agent, emotions may influence decision-making and planning. In a multi-agent, social context, emotions may play an important role in the interaction, especially when humans are involved.

An important feature of sapient agents that we discussed is *learning*. We discussed reasons, opportunities and solutions for learning. For sapient agents, learning transcends the idea of single-task learning by focusing on the whole deliberation cycle, emotional attitudes and the social context. We have defined sapient agents as a generic architecture, based on existing logical, cognitive agent approaches. The main purpose was to highlight various challenges that await when we move from single RMDPs to more general cognitive architectures. Coming back to Section 7.1 we can note the following. First of all, sapient agents have much more complex *mental states* than we have assumed in the previous chapters. In addition to simple beliefs about the current state, sapient agents' minds hold plans, goals, beliefs and intentions for multiple tasks, and their behavior is determined by an iterative and continuous process of *deliberation*. This immediately brings in the second topic discussed in Section 7.1, that of the connection between learning and reasoning. Sapient agents are not designed for one specific task, but instead, use their deliberation cycle to determine their behavior based on various cognitive aspects, in various tasks simultaneously. Because of that, learning and reasoning are naturally integrated, and learning is focused on optimizing the deliberation cycle. This can be done by learning new beliefs, by modifying selection rules for plans and goals, but also by learning to select actions, as in single RMDPs. Sapient agents also integrate declarative and procedural representations, which was the third topic of Section 7.1. Procedural aspects are related to the specific mechanism of the deliberation cycle and the selection of actions. But, all of its beliefs, plans, desires, goals and intentions are declarative, in the sense that the deliberation process has direct access to them and can manipulate them at will.

One line of further research should focus first on formal definitions of the various parts discussed in this section. Formal notions present in formalizations of cognitive agents and agent programming languages should be extended with learning mechanisms, decision-theoretic concepts, probability and emotional attitudes. Also, the connection between *beliefs-desires-intentions* (BDI) logics and MDPs is an interesting direction (Simari and Parsons, 2006). In the previous paragraphs, we have highlighted several challenges for learning in sapient agents, of which some can be solved using existing techniques developed in the context of cognitive agents, yet other ones require new approaches. Based on our description, we can find several aspects for which solutions already have been described in the literature on relational RL. We distinguish five of them here, and discuss how they connect to the structure of sapient agents. In the following section we survey existing approaches that could be applied to sapient agents on all five aspects.

Models. Each dynamic aspect involved in the agent's cognitive architecture adheres to certain rules. The most prominent example is the world itself. When an action is applied, the environment behaves following certain (probabilistic) transition rules. If these rules could be learned from experience, the agent could possibly use them for other tasks, to predict certain events, for (decision-theoretic) planning, or for more informed switching between tasks. Other types of models could predict the effects of selecting other goals to pursue next.

Guidance. For sapient agents, we assume that much of the deliberation cycle has been

specified beforehand. For example, the agent possesses some plans, and some domain knowledge. Additional *heuristics* and *guidance* can help much in deciding what to do next. Guidance can come in the form of a reasonable policy for selecting actions. This way, the behavior of the agent is biased towards interesting, and informative, parts of the state space, such that it wastes less time exploring uninteresting parts of the state space. Other forms of help may come in the form of the selection of (sub-)problems the agent has to solve, in order to supply increasingly more difficult problems to the agent.

Hierarchical Decompositions. Sapient agents will make much use of *decompositions* of problems to select (sub-)goals based on its current mental state. Some solutions for achieving goals may be programmed, but learning them is preferred in general. Sub-goals require a decomposition of a task, such that goals can be reached using a *hierarchy* of behaviors. Learning a *skill* to achieve a sub-goal is very important. Behavior decompositions also add a *modular* structure in the form of *program constraints* to the overall behavior of the agent, which can be exploited in the deliberation cycle by choosing between skills based on the currently selected goal.

Transfer. Because sapient agents are occupied with multiple tasks, it would be useful to *transfer* acquired knowledge from one task to another. Possible targets of such knowledge transfer are skills and models. If the agent has learned some knowledge about one (aspect of a) task, it would be very useful to use that knowledge in a different, but related task. World models are general, and they possibly apply in many related tasks. Skills, once learned and extracted in declarative form, could help the agent solve other tasks, or could be communicated to other agents.

Multiple Agents. When multiple agents are present, the agent can learn all kinds of knowledge and skills from other agents, either by communication or by observation. Formal models such as DEC-POMDPs, multi-agent game theory, and multi-agent epistemic logic could be adapted to first-order logic and used in sapient agents (see Fagin *et al.*, 1996; Kok, 2006; Spaan, 2006, for pointers). FOL approaches have the advantage that other agents can be represented as *objects* in the agent's mental state, and learning and generalization could make use of that.

7.3. A Survey of Hierarchies, Models, Guidance and Transfer

In the previous we have discussed the broader context of solving RMDPs as part of larger, cognitive systems. In this section we survey a number of specific topics in this context that build on the RMDP solution techniques described in the previous three chapters. We first turn to models, guidance, hierarchies and transfer, which are all targeted at computing solutions more efficiently, putting constraints on the learning task, helping the learner, or transferring learned knowledge between different tasks. Afterwards, we briefly mention some decision-theoretic, multi-agent approaches.

7.3.1 Learning World Models

Learning world models is one of the most useful things an agent can do. Transition models embody *knowledge* about the environment that can be exploited in various ways. Such

models can be used for more efficient RL algorithms (see Section 2.6.3), for model-based DP algorithms, and furthermore, they can often be transferred to other, similar environments (see also later in this chapter). There are several approaches that learn general operator models from interaction. Learning is usually limited to simpler formalisms such as STRIPS, because these representations are closer to the typical ILP setting. Learning more complex transition models, for example in situation calculus, results in a learning problem that is still too complex for current approaches.

Usually, *model learning* amounts to learning general operator descriptions, for example STRIPS rules. However, simpler models can already be very useful and we have seen examples throughout Chapter 5. Examples include the partial models used in MARLIE (Croonenborghs *et al.*, 2007b), in QLARC (Croonenborghs *et al.*, 2004), and by Gardiol (2003), and furthermore the abstract models learned for CARCASS by van Otterlo (2004a) and for ICARUS by Langley *et al.* (2004). Another example is the more specialized action model learning employed by Morales (2004a) who learns r -actions using behavioral cloning. An interesting, recent approach was described by Halbritter and Geibel (2007), and is based on graph kernels (see Figure 4.5 in Chapter 4 for an example, and further the KBR approach (Driessens *et al.*, 2006b) in Chapter 5). These kernels can be used to store transition models without the use of logical abstractions such as used in most action formalisms. All model-based approaches that we have described in Chapter 6 on the contrary, take for granted that a complete, logical model is available. A related approach by Mourão *et al.* (2008) is based on *kernel perceptrons* but is restricted to learning (deterministic) *effects* of STRIPS- and ADL-like actions. The approach is developed in a broader context of *robotics* and *vision*, where high-level planners have to be combined with low-level (robotic) vision systems (see Kraft *et al.*, 2008; Petrick *et al.*, 2008).

Learning aspects of (STRIPS) operators from (planning) data is an old problem, and solutions differ in the knowledge they start with (e.g. from scratch, or an incomplete action model), the specific formalism they learn (e.g. STRIPS, ADL, PDDL), whether they can learn in probabilistic environments and whether data comes from e.g. planning procedures and whether states are fully observable. It has proved difficult to assemble a complete list of approaches from the literature. Very early approaches include those based on the *event calculus* and ILP (Sablon and Bruynooghe, 1994), DIFFY-S based on operator *effect* learning (Kadie, 1988), Grant (1996)'s approach for knowledge-based planning, and Shen (1993)'s work on precondition learning. Probably the first explicit approach to operator learning was the work done by Vere (1977, 1978) in the THOTH system. Related approaches that exhibit some form of operator learning in combination with planning and reasoning can be found in the literature on cognitive architectures.

Now let us turn to learning full models. Remember the probabilistic STRIPS action we have defined in the third BLOCKS WORLD experiment in Section 6.3.5. This move action has three different, probabilistic outcomes. The first has the intended effects, whereas the second two represent the cases where the action fails (with probability 0.1) and where the action moves the block onto an unintended block (with probability 0.1). Learning such a description from data involves a number of aspects. First, the logical descriptions of the pre- and post-conditions have to be learned from data. Second, the learning algorithm has to infer how many outcomes an action has. Third, probabilities must be estimated for each outcome of the action. Learning such probabilistic rules from data is a hard task, and falls under the SRL approaches discussed in Section 4.3.3, that learn both logical structures

and parameters from data. In the propositional setting, the earliest approach for learning probabilistic planning operators was described by Oates and Cohen (1996). In Section 3.5 we have seen several additional methods that learn DBN-like transition models from data.

For the first-order case, early approaches by Gil (1994) and Wang (1995) learn *deterministic* operator descriptions from data, by interaction with simulated worlds. Whereas the first aims at *completing* partially incomplete operator models, the latter can start with an empty set of rules. Another example is the approach by Lorenzo and Otero (2000) which is based on logic programming and ILP. Benson (1996) describes the more general TRAIL system that learns *teleoreactive* operators. The representation used is a first-order HORN language, but some *deictic* aspects are incorporated (see Section 4.1.3.3), and furthermore it handles continuous actions and real-valued fluents, and copes with noise and some probabilistic aspects. Jiménez *et al.* (2006) cope differently with probabilistic effects, by ignoring probability and only focusing on learning when (nondeterministic) actions will *succeed* or *fail*, in an incremental setting. Shahaf and Amir (2006)'s SLAF approach does not consider probabilistic effects at all, but instead, focuses on *partially observable* action models, i.e. where some of the effects of actions cannot be observed (see also Amir and Chang, 2008). The related ARMS approach (Wu *et al.*, 2005; Yang *et al.*, 2005, 2007) too handles incomplete state information, and finds action models by posing the learning problem as a SAT problem. The LAMPS approach by Zhuo *et al.* (2007) can be considered an extension of the ARMS system that puts more constraints on the induction problem. It was applied for *software requirement specification*. And whereas the SLAF approach focuses on handling *computational* complexity, the work done by Walsh and Littman (2008) deals with the *sample complexity* of learning (deterministic) STRIPS models. In addition, whereas most systems assume that examples are generated by a planning system, Walsh and Littman also consider the case where a *teacher* is available that can generate optimal plan examples when needed.

When going beyond the deterministic setting, one work addressing the issues of learning *both* structure and parameters of probabilistic, relational planning rules is that by Pasula *et al.* (2004). It assumes that actions will only affect a small number of properties of the world and that the number of different outcomes of one action is small. The action rules are assumed to have a STRIPS-like form, in which a *context* defines an abstract state in which the action is applied and a set of probabilistic *outcomes* defines the changes in the state if the action is applied. Learning rules consists of a three-step greedy search approach. First, a search is performed through the set of rule sets using standard ILP operators. Second, it finds the best set of outcomes, given a context and an action. Third, it learns a probability distribution over sets of outcomes. The learning process is *supervised* as it requires a dataset of state-action-state pairs taken from the domain. As a consequence, the rules are only valid on this set, and care has to be taken that it is representative for the domain. Experiments on BLOCKS WORLDS and logistics domains show the robustness of the approach.

Later, Zettlemoyer *et al.* (2005) extended this approach to handle domains with more complex dynamics (see also Pasula *et al.*, 2007). In many domains the assumption that each action has only a small set of outcomes, does not hold. Some action effects are highly unlikely and therefore hard to model. Consider pushing over a stack of blocks; the number of possible blocks configurations (outcomes) is extremely large. This is solved by using *noise outcomes* for actions, and the exact *structural* result of the action in that

case is ignored. This effectively renders the model *partial* in return for still being able to learn and act effectively. Furthermore, the approach is able to invent new features. New features are expressed in terms of already learned ones, thereby extending the concept language during learning. This is commonly referred to as *predicate invention* in ILP, and has connections with constructive function approximation (see Section 4.1.3.3). Earlier work by Gardiol (2003) who used standard ILP learning on state transitions, exemplified the difficulties of the complex domains that Zettlemyer *et al.* solve.

A second approach to learning first-order, probabilistic operators is introduced by Safaei and Ghassem-Sani (2007). A major difference with the first approach is that Safaei and Ghassem-Sani's approach is incremental, in a similar way as Wang (1995)'s OBSERVER system. The algorithm combines planning (based on RTDP, see Chapter 2) and learning. Trading off acting and exploration provides good opportunities for generating useful learning examples for the induction of operator descriptions. Incrementality is an important difference between Pasula *et al.* (2004)'s approach and the ones by e.g. Safaei and Ghassem-Sani (2007) and Wang (1995). Another difference is that whereas Benson (1996)'s and Wang (1995)'s approaches utilize both positive and negative learning examples, Pasula *et al.* (2004)'s and Gil (1994)'s approaches use only positive ones.

7.3.2 Bias, Guidance and Heuristics

We have described a number of model-based and model-free learning algorithms that can handle the typically huge state spaces that RMDPs induce. Approximations are commonly used to deal effectively with scaling up to even larger domains. However, like in the classical MDP context more methods exist that can help learning, such as smart exploration and learning models of the MDP such that these can be used in learning or planning (see Chapter 2). Many related approaches exist that employ ML for planning approaches (e.g. see Zimmerman and Kambhampati, 2003; Yoon *et al.*, 2005). Here we discuss some approaches that *help* the learner in solving RMDPs.

The most important bias the learner can get is the *language* bias. An RMDP is specified using a basic set of ground atoms in some relational language. But, the language used for representing abstract value functions, policies and models varies much between the methods in Chapters 5 and 6. The more powerful this language is, the more powerful abstractions can be, but the more computational costs are involved in inducing them from data. Some techniques require background knowledge predicates (e.g. TG) whereas others can do without (e.g. KBR). Feature generation techniques, such as we have seen in the preceding two chapters, require such predicates to induce (or deduce) useful formulas to function as first-order features.

In some systems, the samples that are used for learning policies, are generated using prior knowledge about the domain, thereby biasing the learner towards useful samples. Driessens and Džeroski (2002a, 2004) use a hand-coded policy to provide *guidance* to the TG system (see Section 5.3.2), which can be seen as a guided exploration method. The guidance policy generates biased samples that provide (semi)-optimal learning examples that are then treated as normal. If properly mixed with the standard learning mode, guidance speeds up learning. An additional benefit is that it provides feedback in case rewards are sparse (such as with a single goal state which is hard to find with random exploration). Related to guidance is the usage of *behavioral cloning* based on human generated traces, for example by (Morales, 2004a) and Cocora *et al.* (2006), or the use of optimal plans by

Yoon *et al.* (2002).

Fern *et al.* (2004a) (in the context of the LRW-API framework by Fern *et al.* (2003), see Section 5.5.2) do not explore the state space, but instead use exploration on the set of domains instantiations. The systems starts by generating easy instantiations of the given domain and the difficulty of the problems is increased in accordance with the current policy's quality. The instantiations are generated by performing *random walks* in the planning domain, generating a start and goal state. The length of the random walk determines the difficulty of the task. The approach can be seen as a kind of *reward shaping*. Domain sampling was also used by Guestrin *et al.* (2003a).

Lane and Wilson (2005) focus on environments having strong *topological* structure (see also Lane *et al.*, 2007, for additional results in nonstationary environments). Many navigational tasks have a special structure that has a natural relational representation in terms of topological relationships, such as used in *the point n distance to the south of the wall*. By exploiting such knowledge about the domain, policies can be reused or *relocated* by making use of the properties of the underlying metrics and topologies. In Chapter 6 we have mentioned that such topologies can be very helpful in, for example, logistics domains.

7.3.3 Hierarchies

Although hierarchical RL has become mature, such approaches for RMDPs have only recently been explored. An advantage of *relational* HRL is that parameterizations of sub-policies and goals naturally arise, through logical variables. For example, a BLOCKS WORLD task such as $\text{on}(X, Y)$, where X and Y can be instantiated using any two blocks, can be *decomposed* into two tasks. First, all blocks must be removed from X and Y and then X and Y should be moved on top of each other. Note that the first task also consists of two subtasks, supporting even further decomposition. Now, by first learning policies for each of these subtasks, the individual learning problems for each of these subtasks are much simpler. Furthermore, learning such subtasks can be done by any of the model-free algorithms in Chapter 5. Depending on the representation that is used, policies can be structured into hierarchies, facilitating learning in more complex problems. One main advantage that is shared by all HRL methods is that such sub-policies can possibly be reused in other tasks. Whenever the agent encounters a subgoal $\text{on}(X, Y)$ in its current goals, it can reuse the learned skill, and instantiations for X and Y can come from other subtasks in the hierarchy. This can be done, for example, by representing it as an OPTION (see Croonenborghs *et al.*, 2007a, for initial ideas) and inserting it as a skill in the agent's cognitive structure. In Section 3.8 we have discussed several dimensions of HRL and described some of the main approaches. Implementing any of these approaches, such as OPTIONS, or MAXQ, can be done relatively straightforward given that much about *non-hierarchical* model-free and model-based learning is known by now (see Chapters 5 and 6). There are several initial approaches in hierarchical, relational RL (HRRL) and we discuss them briefly.

Driessens and Blockeel (2001) presented an approach that involves some aspects of hierarchical decompositions. The aim is to solve an RMDP with two goals that can be pursued simultaneously, in the computer game DIGGER². Here, the learner has to gather

²DIGGER is, from a learning algorithm point of view, a simple problem that can be easily modeled using propositional representations. Driessens (2004)'s description of the game consists mainly of such features. Results from a student project at our department showed comparable results with a propositional, genetic algorithm.

diamonds while *avoiding monsters*. Both subgoals are first learned in isolation using the Q-RRL system (see Section 5.3.2) and after that, the learner is confronted with the complete task, in which it can use the value functions of both subtasks as background knowledge. This differs from standard HRL in which *sub-policies* are used to generate a hierarchy.

Aycenina (2002) uses the original Q-RRL system to build an HRRL system. A number of subgoals is given and separate policies are learned to achieve them. When learning a more complex task, instantiated sub-policies can be used as new actions. More recently, Roncagliolo and Tadepalli (2004) use batch learning on a set of examples to learn values for a given relational hierarchy (i.e. PIAGET-2). The input to the algorithm consists of tuples of the current state, task, subtask and Q -value. The result is a decision list style Q -value function that generalizes over tasks and subtasks, using piecewise linear value approximations. In the same direction as the two previous approaches, Andersen (2005) presents a more thorough investigation of HRRL using MAXQ hierarchies (see Figure 3.18) adapted to relational representations in the form of logical decision trees. The work uses the Q-RRL framework (see Section 5.3.2) to induce local Q -trees and P -trees, based on a manually constructed hierarchy of subgoals. P -trees can be induced locally, but also globally, for the complete hierarchy. Whereas two out of these three methods learn logical abstractions for subpolicies – and can therefore be classified as PIAGET-3 – they are not constructive learners at the hierarchical level: the hierarchy is supplied to the learner, based on a manual decomposition of the task. Further work to learn such hierarchies automatically would greatly improve the applicability, and many of the algorithms in Section 3.8 can be upgraded to RMDPs.

Two other systems use hybrids of *planning* and learning, which can be considered as generating hierarchical abstractions by planning, and using RL to learn concrete policies for behaviors. The approach by Ryan (2002) (see also Ryan, 2004b) uses a planner to build a high-level task hierarchy. RL is used to learn *teleo-operators* that move from one abstract state in the plan to another. A precondition of such an operator defines the application space of a behavior learned by RL, and the postcondition a local goal for the operator. Grounds and Kudenko (2005) introduce a similar method, differing in that this method operates in terms of STRIPS plans instead of teleoreactive planning. Related to these approaches, Reid and Ryan (2000) employ a similar high-level-versus-low-level division, but focus on learning side-effects of actions using ILP.

7.3.4 Transfer

Learning general knowledge in terms of e.g. models and skills is already very useful in learning a specific task. But, because of the very fact that it is *general* knowledge, it would be useful to extract this knowledge, and use it for different, but similar, problems. In Section 3.3.2 we have hinted at a fifth type of PIAGET-learning, which can be informally stated as *reasoning after solution*, i.e. the extraction of knowledge from a learned solution, forming the opposite of PIAGET-0 learning in which knowledge is inserted in the learning algorithm a priori. The common name for this type of approaches is *transfer learning*.

"A particular topical area of AI research in 2007 is transfer learning: leveraging learned knowledge on a source task to improve learning on a related, but different target task. Transfer learning can pertain to classical learning, but it is particularly appropriate for learning agents that are meant to persist over time,

changing flexibly among tasks and environments. Rather than having to learn each task from scratch, the goal is to take advantage of its past experience to speed up learning.” (Stone, 2007)

It is related to early *speed-up learning* approaches such as EBL (see Chapter 6) in which existing knowledge is transformed into another form that is more suitable for the current task. However, transfer as it is defined now, aims at the more general setting of transferring any kind of (declarative) knowledge to the target domain. A general characterization of successful transfer learning can be measured by the success of learning on the target task. Without transfer, learning the target task would take an effort³ E_t . With transfer learning, the effort is the combined effort of first learning on the source problem (E_s), transferring the knowledge (E_k) and then – with the transferred knowledge – learning the target problem (E'_t). Transfer learning is useful when $E_s + E_k + E'_t \leq E_t$. Transfer is much representation-dependent, and the use of (declarative) FOL formalisms in relational RL offers good opportunities for transfer.

In the more restricted setting of (relational) RL there are several possibilities. For example, the source and target problems can differ in the goal that must be reached, but possibly the transition model and reward model can be transferred. Sometimes a specific state abstraction can be transferred between problems (e.g. Walsh *et al.*, 2006). Or, possibly some knowledge about actions can be transferred, although they can have slightly different effects in the source and target tasks. Sometimes a complete policy can be transferred, for example a *stacking* policy for a BLOCKS WORLD can be transferred between worlds of varying size. Another possibility is to transfer solutions to *subproblems* (e.g. Asadi *et al.*, 2006), for example a sub-policy as in HRL (see Section 3.8). An important distinction in all forms of transfer is between *inductive* and *deductive*. Whereas in the former, an additional learning step is involved which can create new (statistical) inaccuracies, in the latter transfer can be computed more deterministically. Sometimes additional background knowledge (e.g. about the domain) can help. For example, in Chapter 6 we have used additional predicates to compute a generalized policy from a more specific Q -function.

We have seen several examples of transfer in the previous chapters. For example, the inductive extraction of a logical policy as in P -learning or the deductive approach used in REBEL both try to generalize a Q -function such that the resulting policy can be used for other problems. However, most such relational RL techniques rely on the intrinsic opportunities of the domain to be successful. That is, in BLOCKS WORLDS, in most cases a policy that is learned for a task with n blocks will always work for $m > n$ blocks. This is true for the usually employed tasks such as *stack*, *unstack* and even *on(X, Y)*, which are used in many works. In this respect, one has to be careful in defining what exactly counts as transfer in such domains. So far, a few approaches have approached slightly more general notions of transfer in relational RL and here we briefly mention them.

Hierarchical relational RL offers many opportunities to transfer sub-policies to new problems. Croonenborghs *et al.* (2007a) discusses some initial ideas of using the OPTIONS framework (see Section 3.8) for representing such sub-policies in the relational setting. It conveys the same ideas that were implemented in the hierarchical approaches we have discussed in the above. Torrey *et al.* (2006) induce such skills from experience, guided by

³For example, measured in the amount of computation or the number of episodes required.

human advice that consists of indicating which skills must be induced, a mapping between objects in the source and target domains, and optionally some specific advice in generating abstractions for the skills. In a subsequent paper Torrey *et al.* (2007) define *macro actions* as finite-state machines, and address the problem of learning them through ILP from a dataset of experienced episodes, both successful and unsuccessful ones. The method shows the benefits of transferring learned macros, but the specific example domain (simulated robot soccer) did not yet make full use of relational abstraction possibilities.

Some other approaches to transfer in relational RL are extensions of methods described in Chapter 5. All make use of the intrinsic flexibility in their approaches to transfer parts of learned solutions (e.g. value functions) to new problems. Torrey *et al.* (2008) use MLNs (see Section 4.3.3) to store a Q -function structure for a source task which is then used in the early stages of a target task. As in previous work, Torrey *et al.* use a simulated soccer domain. Stracuzzi and Asgharbeygi (2006)'s approach is based on RTD (Asgharbeygi *et al.*, 2006). Remember that the RTD approach uses a linear combination of relational concepts to represent the value function. In their approach, the concepts are the same for the source and target domains, though the domains itself may have a different representation. This amounts to using the same features for both problems, but with different weights. One of the experiments describes a CHESS end-game (*King-Rook-King*) in which first a value function is learned on a 5×5 -board, which is then used for further learning on a 8×8 -board. Ramon *et al.* (2007) use the intrinsic flexibility of the structure-adaptable TGR algorithm to reuse (and further adapt) the structure of the Q -tree from the source domain to the target domain (see also Driessens *et al.*, 2006a). The approach is tested on both a supervised classification problem (*Bongard*, see Chapter 4) with a changing concept and relational RL in BLOCKS WORLDS. Finally, related to these three approaches, García-Durán *et al.* (2008) transfer instance-based policies in a deterministic planning domain.

7.3.5 Multi-Agent Approaches

This book is about the single-agent case. Scaling up towards *multiple* agents brings us to the field of *multi-agent systems* (MAS) (Ferber, 1999; Weiss, 1999). There are many approaches dealing with RL for MAS (e.g. see Abul *et al.*, 2000; Stone and Veloso, 2000; Wiering *et al.*, 2000; Gu and Yang, 2004; Buçoni *et al.*, 2008). Most of these approaches are built on top of single-agent RL algorithms, now dealing with additional topics such as *cooperation* between agents, *communication*, *division of work* and much more. Building on the work on MDPs and POMDPs, more general models involving multiple agents are based on *game theory* and *decentralized* POMDPs (see Kok, 2006, for pointers to the literature). *Logical* approaches are usually based on multi-agent, modal epistemic logics (e.g. see Fagin *et al.*, 1996). The combination of all these fields yields a huge space for further research. Here we briefly mention some approaches that explicitly target RMDPs in a multi-agent setting.

Letia and Precup (2001) report on multiple agents, modeled as *independent reinforcement learners* in the GOLOG extension (Levesque *et al.*, 1997) of situation calculus (Reiter, 2001; McCarthy, 1963). The agents do not communicate, but act in the same environment. GOLOG programs specify initial plans and knowledge about the environment. The complex actions in the GOLOG programs induce a semi-MDP, and learning is performed by model-free RL methods based on the OPTIONS framework (Sutton *et al.*, 1999).

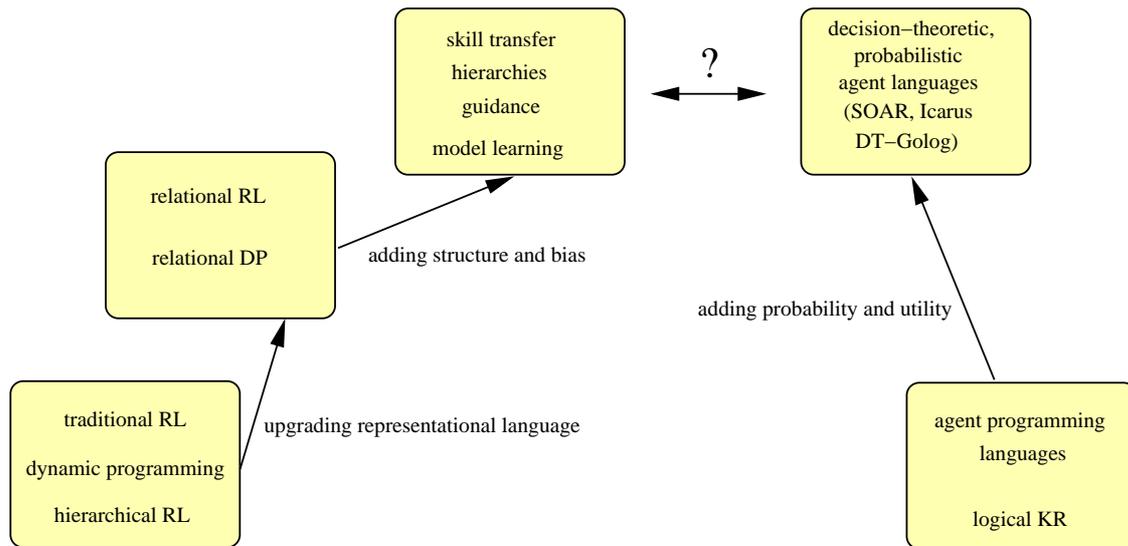


Figure 7.2: Two types of approaches towards general, probabilistic, decision-theoretic cognitive architectures.

Finzi and Lukasiewicz (2004a) introduce GTGOLOG, a *game-theoretic* extension of the GOLOG language. This language integrates explicit agent programming in GOLOG with game-theoretic multi-agent planning in Markov Games. Finzi and Lukasiewicz (2004c) define *relational Markov Games*, which are static structures in stochastic situation calculus, that can be used to abstract over multi-agent RMDPs and to compute *Nash policy pairs*.

Hernandez *et al.* (2004) describe an approach that combines BDI (beliefs, desires, intentions, see more on this Wooldridge, 2002) approaches with learning capabilities (logical decision trees) in a multi-agent context. Finally, Tuyls *et al.* (2005) discuss in a position paper issues about using the system Q-RRL in multi-agent contexts (see also Croonenborghs *et al.*, 2006b, for some experiments in the same setting).

7.4. Discussion

In this chapter we have characterized sapient agents as a means to identify several directions in which to go, in order to scale up to more complex problems and to more general cognitive architectures. Interestingly, we can distinguish two separate, but highly related, movements based on the literature we have discussed (see Figure 7.2).

The first is the particular topic of this book. Starting from the work on propositional RL, we have discussed several abstraction types in Chapter 3 that aim at finding reusable structure in both problems and solutions, and algorithms that exploit them. Using first-order logical KR and reasoning techniques, all these techniques can be upgraded to be applied to first-order domains (see Chapters 4 to 6). A next step, described in the current chapter, is to identify yet more complex structure in the agent's mind, in the form of beliefs, goals, intentions and a deliberation cycle that runs over all these cognitive notions. In Section 7.3 we have surveyed several areas in which techniques have been developed that can help learning approaches in these cognitive architectures. The main thread in the left side of Figure 7.2 is one of finding more structure, finding more complex patterns, and scaling up adaptive decision making architectures.

A second direction we can identify, comes from the work on agent programming languages and cognitive architectures (see the right side of Figure 7.2). There are several recent advances in adding probability and decision-theoretic concepts into such languages. With the increased insight into efficient reasoning and learning techniques for probabilistic and decision-theoretic problems in first-order domains, such languages can be made richer, and some approaches have done so already. The starting point of this direction is a lot more advanced than the approaches in the left side of the figure. Existing logical systems support various forms of reasoning, *ontologies*, *sensing actions*, *modal constructs* and considerably more complex cognitive structures and functional processes that operate on them.

Both directions will meet at some point, in the figure denoted by the question mark. This means that there are many possibilities for cross-fertilization between the two directions. Some are right in front of us, for example the combined study of modal constructs and the solution of POMDPs, whereas others, such as the incorporation of *concept building* and *grounded knowledge* require more insights and efforts.

CHAPTER 8

Conclusions and Future Directions

In the last chapter of this book, we reflect on what has been accomplished. We draw general conclusions, summarize the main argument of the book and identify several important dimensions of learning sequential decision making in first-order domains. A substantial part of this chapter is concerned with identifying directions for further research.

THE WORLD DOES NOT APPEAR to be just a formless assemblage of myriad states as it would if mind did not understand its structure. Finding exploitable structure in learning sequential decision making has been the main topic of this book. More specifically, we have focused on *first-order* structure, where the world consists of objects. Relational RL aims at tackling one of the central open questions in ML and AI, namely reasoning, acting and learning in richly structured, first-order probabilistic worlds. In the first years after the seminal work by Džeroski *et al.* (1998) some techniques were introduced, but in recent years, the field has rapidly expanded by introducing a whole range of methods for model-based and model-free solution techniques. A central issue is how to trade-off the complexity of the logical abstractions and useful approximations to be used in increasingly larger problems. At the heart of all challenges arising in relational RL lies the *trade-off between exploration and exploitation of logical abstraction levels*. To accumulate a lot of reward, the learning system must *exploit* the best experienced abstraction level. However, to discover better action selections for the future, it must *explore* new abstraction levels.

Besides this book, there are at least six other recent PhD. theses that are devoted to the topic of learning sequential making in first-order domains. The first was mainly based on the algorithms Q-RRL, TG, RIB and KBR and was written by Driessens (2004), and descriptions of these techniques were given in Chapter 5. The second is based on the work by Sanner and Boutilier which we have described in Chapter 6 and has recently been written by Sanner (2008a). The third was written by Wang (2007) and we have described many of its contributions in Chapter 6. The fourth dealt mainly with the evolutionary approach FOXCS for model-free relational RL and was written by Mellor (2007). The fifth was written by Gardiol (2007) and deals with relational envelope-based planning. The sixth thesis was written by Yoon (2006) and is about learning control knowledge.

The coming years, the field should continue to expand by upgrading more solution techniques from the classical RL framework. However, theoretical advances are needed to be able to understand better the various interactions between logic, utility and probability. Furthermore, a direction that will prove to be fruitful for general artificial AI is connecting agents and powerful logical reasoning systems with relational RL. Some of the techniques in this book have shown considerable progress in this respect, and it is interesting, and vital, that more work studies the interaction between reasoning and learning. Making use of lots of available knowledge breaks with the *tabula rasa* style of early work in RL and will bring the construction of even more intelligent agents closer.

Even though we advocate the use of first-order formalisms for RL problems, – mainly because there are many good reasons for doing so (see Section 4.1) – it is not to say that all problems should be modeled as such. In the general ML setting, *collective classification* (Taskar *et al.*, 2002) was one of the breakthrough applications for showing the benefits of relational representations. A general guideline is that the more relational patterns exist between entities that are present in the data, the more likely it becomes that relational representations pay off. The vast majority in statistical ML algorithms has focused on so-called flat data, i.e. indentially-structured data that is typically assumed to be independent and identically distributed (IID). On the other hand, many real-world problems possess highly relational structure, such as cross-citations in scientific papers, hyperlinks in webpages, and social networks. For relational RL there are several problems that require a first-order representation, most notably the BLOCKS WORLD. Many other application domains have been used in relational RL research. Sometimes representations for such domains can be *propositionalized* into a flat representation, treating it as if it were propositional. However, this throws away much of the relational dependencies between entities. A similar phenomenon can be found in propositional representations, in the form of *parity problems*. Thornton (1996, 2000) shows that, while such problems may be learned using statistical learning algorithms, the end result may be one in which each single datapoint must be represented separately, preventing any gain by generalization.

8.1. Conclusions and Reflections

The first part of this chapter will reflect on the material in this book. First, we summarize our main line of argumentation in the Chapters 2 to 7. After that, we highlight our contributions to the field and list some of the main dimensions of relational RL.

8.1.1 Main Argument

Our first line of questioning in Chapter 1 asked how adaptive behavior can be modeled using MDPs and which types of algorithms exist to solve them. We have dealt with these questions extensively in Chapters 2 and 3.

The first of these chapters has given an overview of the main types of models, algorithms, and efficient extensions. It describes the setting of the robot CANTOR introduced in the first chapter. We have restricted our discussion in this book to the Markovian setting, but in Section 2.7 we have highlighted some of the conceptual and technical problems of more general models such as POMDPs. It turns out that most algorithms for solving MDPs work according to a principle called *generalized policy iteration* (GPI), in which computation of value functions is alternated with computation of policies.

A — Generalized Policy Iteration (GPI)

In each iteration of GPI the current policy is evaluated and this results in the estimation of a value function. By looking at the value function, possible improvements to the current policy can be found and a new, improved policy can be computed, and so on. A crucial distinction between algorithms is based on what they assume.

B — Model-Based versus Model-Free

Model-based algorithms assume that the full transition and reward model are known, and from that, optimal solutions can be computed. On the other hand, *model-free* algorithms do not assume this knowledge, and instead, have to *interact* with the environment to obtain *samples* of states, actions, and resulting transitions and rewards, and have to base their value function on these estimates. This creates two entirely different classes of algorithms, though both follow the general principle of GPI.

The second half of the answer to the first set of guiding questions has been given in Chapter 3. We have identified a number of issues with the MDP framework in Chapter 2. Most importantly, the incapability to identify and exploit *structure* in problems and solutions is an obstacle to employ the MDP framework for any realistic problem. When problems become bigger, it is no longer possible to store, update, and explore the complete MDP, transition functions, value functions and policies. A general solution is to use a propositional, feature-based representation of the world, and employ *abstraction* and *generalization* to exploit the structure hidden inside them. This is the setting for the robot BOOLE, introduced in the first chapter.

C — Abstraction and Generalization

Abstraction is the ability to abstract away from irrelevant details unnecessary for the current task, or current decision, whereas generalization is the general process of obtaining such abstractions. Structure in MDPs and their solutions can be found in a number of ways, and we have distinguished five types in Chapter 3. For model-based methods, the challenge is to identify structure in the problem and maintain and exploit that while computing optimal solutions. For model-free algorithms, the challenge is to generalize the samples obtained by interacting with the environment.

D — Five Main Abstraction Types

Each of the five types focuses on a different element of the MDP framework. Whereas the first four types basically exploit the propositional structure of states, the last one exploits structure in the MDP transition graph.

✓ **State Spaces.** By identifying *groups* of states that are similar, so-called *abstract states* can be constructed, which can then be used to treat different states as if they were the same in learning. Each time something about a specific state has become known, it is automatically generalized to many other states (see further Section 3.4).

✓ **Factored MDPs.** Factored MDPs are highly structured representations of MDPs. For example, compact representations of transition functions and reward functions using Bayesian networks and decision trees can be obtained by making use of the propositional structure of states and the fact that the values of individual features often depend only on a limited context. Most algorithms are model-based, and they maintain, and exploit, the structured representations throughout computation (see further Section 3.5).

✓ **Value Function Approximation.** By generalizing over the mapping from states to values (or state-action pairs to values), value function approximation techniques can represent value functions with a complexity that only depends on the set of parameters of the approximation architecture, instead of the number of states in the MDPs. Many are based on neural networks, linear value approximation, or decision trees (see Section 3.6).

✓ **Policy Search.** Some algorithms focus on abstraction of the policy itself. Instead of learning value functions, they search for policies directly. Based on the samples obtained while interacting with the environment, they induce policy representations, which are compact representations that exploit propositional structure in the mapping from states to actions (see further Section 3.7).

✓ **Hierarchical Decompositions.** In many MDPs it is possible to find structure in the action transition structure of optimal solutions. The policy itself can consist of multiple, modular subpolicies that each solve a particular subtask in the MDP, for example to get from one set of states to another. Hierarchical decompositions can be seen as imposing a structure on the policy, or decomposing an MDP into multiple, simpler decision problems, again a kind of MDPs (see further Section 3.8).

For each of these abstraction types, we have described many ways to exploit the specific type of structure. As it turns out, MDP solution algorithms that employ abstraction and generalization, can be characterized using an extended version of the GPI principle. In Section 3.3.2 we have defined the PIAGET principle that makes the use of abstraction and generalization explicit. In such contexts we have to distinguish between *structures* and *parameters*. The structures are the abstractions used to exploit structure in the MDP and its solution. It may be a neural network that represents the value function mapping, or it may be a set of logical formulas that partitions the set of states. The parameters can be values or transition probabilities, now defined over the abstractions (e.g. a set of abstract states). But, parameters can also denote *internal* parameters that have nothing to do with the MDP per se, but belong to a generalization architecture, for example, the weights in a neural network architecture. Having said this, we have identified four ways of learning either structures or parameters.

E — PIAGET: Four Levels of Learning

✓ **PIAGET-0: Generating Structural Components before Learning.** The structure of abstractions can be computed before the MDP is solved, for example, by computing an abstract state space or propositional features from the domain model or from sampled traces through the MDP. In this way, the actual solution algorithm starts with a more compact model to begin with.

✓ **PIAGET-1: Parameter Learning.** Some algorithms may fix the structural components beforehand (either by supplying it or using a PIAGET-0 learning phase) and learn values, transition probabilities, rewards and actions based on this abstraction level.

✓ **PIAGET-2: Parameter Learning with Internal Parameters.** In addition to PIAGET-1, the algorithm may also have to learn or compute values that are not directly associated with the MDP itself, but more with the internal parameters of the abstraction, for example, the slopes in an RBF network.

✓ **PIAGET-3: Combined Structure and Parameter Learning.** The most advanced type of algorithms combines both structure learning and parameter learning in one algorithm. This makes such algorithms very flexible, but also more complicated. Structures influence which kinds of values are learned, and vice versa. Balancing both computational processes is a delicate operation.

Note that, as with GPI, these types of learning may be employed with either model-based or model-free techniques. By combining the five types of abstraction (**D**) and the four PIAGET-levels (**E**) we obtain a reasonably complete picture of propositional MDPs solution techniques with abstraction and generalization. It is this structure that we impose on the field that will help us to utilize the knowledge gained in propositional domains, in *first-order* domains. In other words, as we have discussed in Chapter 1, we can reuse much of our robot BOOLE when building one such as FREGE.

Now in order to go over to defining the context for FREGE, we need *first-order* representations for MDPs. Our second set of questions in Chapter 1 was basically about how to do that, and what can be learned from the propositional setting. For extending the MDP framework, as well as the abstraction types (**D**) and the PIAGET-levels (**E**), to the first-order case, we need reasoning, learning and acting capabilities.

F — First-Order Tools

The three tools are described in Chapter 4.

✓ **First-Order Knowledge Representation and Reasoning.** The most fundamental change is that we go from BOOLE's representation (see Figure 1.5), which is based on propositional logic, to FREGE's representation, which is based on *objects* and *relations*. We have introduced several FOL formalisms that can represent such worlds and abstractions over them, and reason about them using general logical languages (see Section 4.2).

✓ **First-Order Induction and Generalization.** For generalization purposes, we have described ILP and probabilistic extensions in the form of SRL (see Section 4.3).

✓ **First-Order Action Languages.** In order to specify the dynamics of first-order worlds, we have described several formalisms that use first-order languages to characterize the effects of actions on objects and relations. In addition, we have highlighted some of the problems, such as the ramification and frame problems (see Section 4.4)

There are many different types of first-order formalisms for any of these tasks. Based on these languages, we have defined RMDPs (see Definition 4.1.1) as regular MDPs where the states are represented in terms of objects and relations. Furthermore, we have defined the generic FORM formalism (see Definition 4.5.5) for abstraction over the RMDP's state space, value functions and so on.

G — RMDPs and FORMs

The translation step from propositional MDPs to RMDPs now consists of replacing state representations, action representations, abstraction and generalization, all by their first-order versions (**F**). Furthermore, the PIAGET principle (**E**) for learning can directly be

transferred to the first-order setting, where structures now correspond to first-order abstractions.

H — PIAGET in First-Order Domains

There is, however, one fundamental difference. In the first-order setting, abstractions created using logical languages may apply to a whole range of related problems, a so-called *family* of RMDPs. This means that learning can – in principle – be done for multiple problems at the same time, something which is not possible in the propositional setting.

Based on the PIAGET principle in first-order domains (**H**), the first-order tools (**F**), and the basic distinction between model-free and model-based learning (**B**), we describe at length all first-order approaches to model-free learning in Chapter 5 and model-based learning in Chapter 6. Having defined a suitable framework for the field of propositional MDPs, we can clearly see that the field of relational RL has largely the exact same structure as its propositional counterpart. This observation is one of the main accomplishments of this book. Table 3.1 at the end of Chapter 3 makes this correspondence explicit. There are many examples where the correspondence is quite direct. For example, the original Q-RRL approach (Džeroski *et al.*, 1998) upgrades the propositional *G*-algorithm, RUTREE (Dabney and McGovern, 2007) upgrades the UTREE algorithm (McCallum, 1996), and Morales (2003)’s work upgrades work on propositional state abstraction (Singh *et al.*, 1995). Our framework for relational RL is similar in spirit to the work done by van Laer (2002) who defines an upgrade methodology for general ML, mostly the classification setting in ILP. Furthermore, it is similar to efforts in SRL (see Section 4.3.3) where first-order upgrades of probabilistic techniques such as Bayesian networks are developed. Our framework furthermore connects the two strands of developments in both RL and general ML described in the introduction chapter.

There are a number of benefits of our framework. First, it enables us to better understand how first-order techniques work, by looking at their propositional counterparts. Second, it makes it easier to exploit existing knowledge about propositional algorithms and their behavior in their first-order counterparts. Third, it allows for existing (theoretical) results to be carried over to first order algorithms, sometimes without much change. Fourth, it allows us to identify future possibilities and challenges (see Section 8.2).

Finally, in Chapter 7 we have looked at various topics that go beyond state and action abstraction, and focus on the search for reusable structure in models, hierarchies, and cognitive architectures. This chapter is aimed at describing the implications of representations and algorithms for first-order MDPs, answering part of our third set of questions in Chapter 1. We have looked at concepts and techniques that either use RMDP modeling and solution techniques in more complex systems (such as cognitive architectures and hierarchical decompositions), or are aimed at *assisting* the learner. Examples of the latter are the induction and exploitation of *transition models*, the possibilities of *transfer* of learned structures to other tasks, and the use of *heuristics* and *guidance* to enable more efficient RMDP solutions. The remainder of the third set of questions is answered in the current chapter, especially in Section 8.2 where we outline possible directions for future research.

8.1.2 Contributions

This book makes several contributions to the science and engineering of MDPs in first-order domains. Our conceptual contribution consists of the framework described in the

previous section, and our technical contribution consists of several new representations and algorithms. Furthermore, we have surveyed the complete field of relational RL.

A Conceptual Framework for the Use of Knowledge Representation in MDPs. In the previous section we have outlined the main argument of this book. By identifying several abstraction types (**D**) and learning settings for structures and parameters (**E**) we have laid bare a structure that is present in the propositional MDP setting. Upgrading key tools (**F**), representations for first-order MDPs (**G**) and the PIAGET principle, to first-order domains (**H**), has made it possible to impose the very same structure on the first-order MDP setting. This framework enables us to understand the first-order setting (Chapters 5 to 7) by looking at the propositional setting (Chapters 2 and 3) through the lens of first-order techniques provided in Chapter 4.

The **intensional dynamic programming** (IDP) paradigm we have developed in Chapter 6 follows the same pattern as our conceptual framework, but is much more specific about exact DP algorithms. In fact, IDP transcends the complete range of representations, from atomic states to first-order state representations. The close connections between DP algorithms in various representational systems is made explicit and exemplified by the development of the first-order DP algorithm REBEL.

In addition, in Section 3.9 we have described a new application for RL. **Fingerprint recognition** is not an immediate candidate for a solution involving sequential decision making. Usually, a combination of smart feature engineering and pattern recognition is used to match a fingerprint against known prints in a database. We have used RL to learn paths through the fingerprint image, and use the resulting paths to find characteristic features (i.e. *minutiae*) for a possible match. In the context of our framework, we have used this application to highlight the benefits of propositional generalization in RL.

A Complete Survey of the Field of Relational RL. As promised at the end of Chapter 1 we have surveyed all works in the field of relational RL. This survey consists of many subdivisions, along several dimensions such as "model-based vs. model-free", "logical partitions vs. feature-based representations", "value-based vs. policy-based", and "approximate vs. exact" and many more.

Several New Techniques for Relational RL. A considerable part of this book is devoted to describing new formalisms and algorithms for relational RL. We have made contributions to three of the main subfields in relational RL. The CARCASS formalism (see Section 5.2) is a representation for model-free, value-based, relational RL over fixed abstraction levels. Another model-free method that was introduced in this book, is GREY, an evolutionary policy search method. In addition to these model-free techniques, we have introduced REBEL in Chapter 6. REBEL was the first implemented first-order DP algorithm described in the literature.

8.1.3 Dimensions of First-Order MDPs and Solution Algorithms

Based on our findings in this book, our theoretical and experimental work, and the survey we have provided, we can distinguish several important dimensions of first-order MDP representations and algorithms.

Models. Probabilistic transition models of the MDP are very important. For starters, their availability determines whether (exact) solutions can be computed using this model

(as in Chapter 6), or that interaction is needed to obtain samples of the MDP's behavior (as in Chapter 5). Models in the first-order setting are more complex than those in the propositional setting. In the latter, models can often be represented by, for example, a DBN over a relatively small set of domain features (see Section 3.5). In the first-order setting such features correspond to ground, relational atoms, of which there are usually too many to model explicitly. However, first-order languages offer powerful constructs such as variables and quantifiers, which can be used for compact specifications, for example using STRIPS, PDDL, or situation calculus (see Section 4.4). Transition models can be employed for first-order DP algorithms that use them to compute Bellman backups on the logical abstraction level (see Chapter 6). However, note that these algorithms assume that the models are complete and correct. Learning such models is a difficult task (see Section 7.3.1), and usually they are specified by hand. Approximate models on the other hand, can still be useful, and are easier to learn. Chapter 5 has described several algorithms that employ partial, learned models to make value-based learning more efficient.

Abstraction and Generalization. First-order abstraction is often performed by the use of a *language*. The expressivity of such a language determines how compact and expressive abstractions can be, but it also determines the complexity of reasoning and learning algorithms. Concept languages are used by several systems and they are particularly useful for policy representations. Model-based techniques often use considerably more powerful languages, such as situation calculus and fluent calculus. For model-free techniques, often much simpler formalisms are used because of the complexity of learning in more expressive logics (although, see Cole *et al.*, 2003, for an exception in HOL). Existentially quantified, conjunctive languages are often used because they enable efficient reasoning, and because such expressions can be learned using ILP techniques (see Section 4.3.2). Note that representation languages in the first-order setting require increased efforts in supplying additional language biases and a suitable generalization space (e.g. a generality order such as using θ -subsumption), especially when abstractions must be learned from data. This is different from the propositional setting where many methods rely on generalization in the n -dimensional (Euclidean) feature space.

In addition to logical formulas for abstraction over, for example, sets of states, we have seen several other types of representations and generalization techniques. Deictic representations, for example, embed relational aspects in an essentially propositional representation (see Section 4.1.3.3). Furthermore, we have seen several examples of *first-order features* that can be used in linear value function approximations, both in the model-based and the model-free settings. Still largely unexplored are first-order *distances* and *kernels*, until now only used by the RIB, KBR and TRENDI techniques (see Chapter 5). These techniques are very promising, because – in essence – a perfect distance function between examples is all that is needed for any ML algorithm. Still, current techniques in the first-order setting are too computationally expensive and often domain-dependent. An additional aspect is that the results of learning are not declarative and relatively incomprehensible when compared to crisp, logical abstractions.

One solution that has been used by both TRENDI and RUTREE, and which is particularly useful for value functions, is to have a combination of different representations. Logical abstractions can be used for a coarse partitioning of the value function, whereas more fine-grained distinctions can be made using an instance-based representation in each partition. The result is a representation that is more flexible than purely logical abstrac-

tions, yet more comprehensible than purely kernel or distance based approaches.

Structure versus Parameters. The interplay between learning structures and learning parameters is highly relevant in the first-order setting. Only some approaches deal with parameter-only learning (PIAGET-1 and PIAGET-2) based on a priori induced abstractions (i.e., PIAGET-0), or on user-supplied ones as in CARCASS (see for example Section 5.3.1). Most algorithms deal with both problems simultaneously (i.e. PIAGET-3). A challenge in the model-free setting is that the samples used for structural induction algorithms constantly change when the learner gets better at its task. For RL tasks, structures should be flexible to cope with this problem, though this requires algorithms that can adapt their structural representation online, which has only been explored by some recent methods such as RUTREE and TGR (see Chapter 5). A commonly used, alternative solution is to use an iterative process in which sampling and structural induction are interleaved, and where complete new structures are induced in each iteration, for example, as in the LRW-API approach.

Interestingly, the simultaneous learning of both parameters and structures creates new opportunities for sampling and exploration. Whereas in most methods the sampling process is focused on getting learning samples for the next structural induction step, some techniques use sampling in the structural induction process itself. For example, the RUTREE method uses stochastic sampling to generate new logical structures for the value function. Related to this is the search technique used by the FOVI method in Chapter 6 that is used to limit the number of structures that have to be computed. In general, and in line with our discussion of exploring abstraction levels in Section 4.5.2.2, there are two types of exploration processes. One is concerned with sampling based on current structures, whereas the other is focused on the exploration of the space of structures itself.

Deduction versus Induction. Because of the logical context, there is a natural distinction between logical reasoning (i.e. deduction) and learning (i.e. induction). All techniques in the first-order setting employ some limited amount of reasoning, for example to determine whether a state is covered by a logical abstraction. For the rest, model-free algorithms are entirely focused on inductive processes. Model-based techniques on the other hand, are usually based on deduction. For exact methods, the optimal value function and policy can directly be deduced from the domain specification, using decision-theoretic regression. Methods such as SDP and REBEL use general first-order, probabilistic reasoning to derive an optimal value function from the initial reward function definition, using the domain model as a kind of logical axioms. Mixed approaches of deduction and induction are still largely unexplored. An interesting exception is the approach by Gretton and Thiébaux (2004a) which uses deduction to generate logical structures that cover relevant parts of the state space, whereas induction is used to learn a value function using these structures. For the derivation of policy structures from value function structures both induction and deduction can be used. Whereas some approaches use deduction, such as the SDP approach, most others use a P -learning approach where samples obtained from the current value function are used to induce a policy structure.

Value Functions versus Policies. The choice for learning a value function or a policy plays an important role in model-free methods. It is too early to conclude which type of learning to prefer, but the current best performing method is policy-based (LRW-API). Policy-based methods have the advantage that their search space is much smaller, and that

they are easier to apply to complete families of RMDPs. Value-based methods are affected by the domain size, and require often highly fine-grained representations. In Chapter 6 we have shown that value functions can be infinite when the state space is infinite, but a policy structure derived from this value function can be compact and finite. Most model-based methods focus on value functions, where – because of the availability of a model – exact values can be computed. In the model-free setting values must be estimated, and the challenge is to distinguish between inaccuracies that are due to an abstraction level that is still too coarse and inaccuracies that are caused by the stochasticity in the RMDP itself. A related framework that should be explored in relation to the material in this book is *preference handling* in the relational setting (see Brafman, 2008).

Exact versus Approximate. Even though logical formalisms can yield powerful abstractions over RMDPs, many problems are still too large to be solved exactly. There are several model-based techniques such as REBEL that can compute exact value functions and optimal policies. However, their computational complexity limits the size of the problems they can handle. Many approximate versions of such algorithms have appeared that approximate the value function using first-order features. The quality of these features determines how well they can approximate the value function. Some initial approaches have appeared that induce these features from the model description, from statistics obtained from DP algorithms, or from interaction with the environment. Model-free algorithms, on the other hand, all focus on approximate solutions because their structures are computed from samples obtained while interacting with the environment. The combined aspects of the quality of the approximation architecture and the particular sampling process determines how well they can approximate the value function. Model-free policy-based approaches are equally affected by the quality of their samples, though in a different way. They have to assess the quality of the complete policy based on the samples they get when applying that policy in the environment. Some compute values for particular samples online (e.g. LRW-API that uses policy rollout) whereas others employ a holistic policy evaluation through a fitness function (e.g. GREY).

Having described our main conclusions, the framework and survey that are contained in this book, and the aforementioned dimensions of the field, we can defend several points of view about the field of relational RL.

Relational RL: New Paradigm or Interdisciplinary Field? Relational RL deals with learning sequential decision making in first-order, probabilistic worlds. It combines logical representations, probabilistic reasoning, decision-theoretic reasoning and active learning in one coherent framework. Its problems are relatively well-defined and domains where it can possibly be applied are numerous. In this respect, one could defend the position that relational RL is a new paradigm. On the other hand, based on the historical developments in the fields of ML, RL and planning (see Chapter 1) as well as the four viewpoints described at the end of Chapter 4, it is equally valid to see relational RL as an interdisciplinary field that combines techniques from different fields to solve complex problems. In addition, based on our main argument of this book described in Section 8.1.1, it can also be seen as a natural extension of RL, building on existing knowledge. It largely depends on your background and your point of view how to see the field of relational RL. Nevertheless, there are numerous ways to explore new representations and better algorithms. In the

following section we present a representative set of possible further research directions, though we do not – and could not – claim that this list is complete.

8.2. Future Challenges

Because relational RL is a young field, there are numerous possibilities for further research. Here, we present three general types of future directions. The first consists of upgrading existing propositional techniques that have not yet been employed in the first-order setting. The second direction is about extensions to the methods developed in this book. The third direction consists of a list of general topics that are still (inadequately) explored.

8.2.1 Upgrading the Complete Spectrum of RL Methods

The most obvious of all challenges presents itself immediately. The field of RL has presented a large body of research, consisting of a large number of methods. We have described some of the main approaches in Chapters 2 and 3, but there are many more. A continuing challenge is to upgrade more methods to first-order domains. Just to give some examples, the use of neural networks, Bayesian networks, hidden Markov models, decision trees and other architectures is widespread in propositional RL. Not many approaches in relational RL utilize first-order upgrades of such methods yet, although there are many today (see Section 4.3.3). Furthermore, model-minimization approaches and various forms of adaptive resolution RL can be adapted to the first-order setting by developing suitable state splitting and merging operators based on first-order logic. There are also many opportunities for decision-theoretic planning (e.g. see also Sanner, 2008b). Hierarchical approaches based on SMDPs are largely unexplored in the first-order setting. Some work on factored MDPs has been described in Chapter 6, but there is much room for efficient representations and algorithms based on first-order trees, ADDs and BDDs, in various languages. Many other types of algorithms in propositional RL are based on local state space approximations, mainly using *distance* functions. Work on efficiently computing distances between first-order state representations is still a largely unexplored area. Other, more specific techniques that have not received much attention in relational RL are methods based on average reward, actor-critic algorithms, classifier systems, monte carlo simulation, exploration, and asynchronous DP, but there are many more.

8.2.2 Techniques Developed in this Book

In this book we have developed several new techniques. The fingerprint recognition application in Chapter 3 can be extended in various ways, but these lie outside the scope of this book. Extensions of sapient agents as defined in Chapter 7 will be discussed later in this chapter. Here we list some possible extensions to CARCASS, GREY and REBEL.

CARCASS The CARCASS representation provides an elegant and general abstraction for RMDPs. One of its drawbacks is that the abstraction must be constructed manually. It would be desirable to automate this process, and learn such abstractions from interaction with the environment as in AMBIL. This could be done in a top-down fashion, by refining the abstraction based on statistics such as variance in the TG approach. Another way is to find clusters of similar states and use bottom-up generalization techniques based on lggs to find abstract state descriptions. Either Q -learning or PS could be used to evaluate

the current abstraction layer and to find possible refinements. One of the difficulties is to connect a state to multiple actions in a syntactic way, using variables. This presents a delicate problem, because state descriptions have to provide all necessary preconditions for multiple actions at the same time. Restricting the number of actions for each abstract state to one might make the induction of states easier. Other interesting directions are to extend CARCASSs to Q -learning for relational SMDPs, and to employ different kinds of learning algorithms. For example, the use of eligibility traces might make it easier to cope with the partial observability introduced by the abstractions.

GREY The algorithmic of GREY is basically equivalent to the early genetic algorithms. In the mean time, the field of evolutionary computation has produced a vast number of techniques, and based on this, GREY could be extended in many different directions. Different selection techniques, fitness functions, recombination and mutation operators, and various heuristics could be studied. Also, the use of more established evolutionary ILP approaches (Divina, 2006) could be adapted to the same setting as GEY and be used for policy search in relational RL. One specific aspect of study would need to be the difference between deterministic, decision list policies and probabilistic versions.

REBEL Future directions for REBEL, or more general, for first-order IDP, can be found by looking at the range of methods proposed so far. First of all, there are many (probabilistic) action languages (e.g. see Chapter 4) that might be used in this setting. Choosing yet another language might be advantageous because of specific properties of such languages that are useful in IDP. Also, the use of a full first-order logic such as situation calculus in a fully automatic fashion for exact model-based algorithms would be interesting because of its expressivity (e.g. capability of expressing universally quantified goals). Language features that are desired for many realistic applications include sum and count aggregators, explicit quantity and possibly concurrent actions. Orthogonal to increasingly more complex languages is the identification of various sources of *structure* in decision-theoretic problems. Factored models such as investigated by Sanner and Boutilier (2007) make use of this structure in algorithms, and by that, show that increased language power and possibilities for exploiting this structure in representations and algorithms, go hand-in-hand. More specifically for REBEL, extensions with additional background knowledge would be useful to study efficient handling of ramifications in first-order decision-theoretic regression. Furthermore, more work on the use of tabling and policy extraction would enhance the capability of REBEL to cope with more complex problems efficiently.

Feature-based IDP methods are quite well developed, and some theoretical aspects (e.g. performance bounds) have been reported. Many opportunities exist in the direction of the generation of features for specific problem (families). Various inductive (ILP) algorithms can be used for this task. The real challenges lie in the identification of what it means to be a 'good' feature in a certain domain. Many opportunities exist in the *combination* of exact algorithms with approximate aspects. We have discussed some approaches, for example using search and using induction for value function approximation, but the literature on (propositional) factored MDP contains many algorithms that may be upgraded to the first-order case. A last direction should be concerned with theoretical properties of first-order IDP systems, such as convergence properties and computational complexity. When we go beyond the Markov property, the field is wide open to approaches for first-order POMDPs, SMDPs and predictive state representations.

8.2.3 Representational Aspects

As concerns the representational dimension in relational RL, an obvious research direction is to employ other FOL-based representation languages. Research is needed on general languages, but also on the advantages of concept languages and deictic representations, because these offer some specific advantages such as the ability to avoid unnecessary naming of objects. Incorporating specific features into languages, for example to specify topologies, quantities and continuous values would greatly enhance the applicability of relational RL methods, provided that algorithms can make efficient use of them (for example, using tractable inconsistency detection algorithms for numeric values). An additional research topic is to identify more structure such as used in factored representations of RMDPs (Sanner and Boutilier, 2007, see also Section 3.5).

An important direction for future work is studying better formalisms for *approximate* representations in relational RL. Crisp logical abstractions are useful and comprehensible, but especially for value functions we need smooth approximation architectures for first-order interpretations. In the first-order setting, there is no such thing as a black-box type MLP architecture that can learn any desired mapping from states to values. Some useful hybrid representations (e.g. in RUTREE and TRENDI) use a combination of logical abstraction and instance-based regression. Kernels for first-order interpretations can provide the same kind of smooth approximations and more research is needed on computationally efficient algorithms for computing them. An alternative is to use first-order features. We have surveyed several successful techniques based on such features in Chapters 5 and 6. We have discussed both model-based and model-free algorithms that use them, but much more research is needed on suitable *weighting schemes* in the first-order setting, as well as algorithms for generating them, either from the domain model or from interaction experience (see Sections 6.5.2.2 and 5.3.2). Many of such feature generation algorithms are domain-dependent. It would be very interesting to upgrade Mahadevan and Maggioni (2007)'s approach to first-order domains and extract general basis functions that can be employed in relational RL algorithms. Another idea is to employ link mining and graph mining algorithms from SRL to find useful abstractions over the RMDP's state space. In addition, the definition of an *interface layer* (Domingos, 2008) consisting of first-order representations and probability for AI, might be very useful for the work in relational RL too. In all approaches that generate logical abstractions automatically, *predicate invention* is a very desirable feature because it adds even further flexibility and autonomy.

Yet another direction is the use of *recursion* in representations. Recursion is an integral part in defining general PROLOG programs, but automatically inducing them from data is a difficult task. However, they could be very useful for sequential decision making tasks. Consider the *Towers of Hanoi* problem. An optimal solution can be encoded compactly using a recursive program. If such definitions could be learned automatically, they could provide a basis for compact, hierarchical and recursive *skills*.

Comprehensibility of learned abstractions is one of the promises of relational RL. Although relational rules can be more comprehensible than e.g. the weights of a neural network, there are some trade-offs: consider the following abstraction in the BLOCKS WORLD:

- (1) $\text{on}(a, A), \text{cl}(a), \text{on}(b, B), \text{on}(X, b), \text{cl}(X), X \neq a$
- (2) $\text{on}(a, A), \text{on}(X, a), \text{cl}(X), \text{on}(b, B), \text{cl}(b), X \neq b$
- (3) $\text{on}(a, A), \text{on}(b, a), \text{cl}(b)$
- (4) $\text{on}(a, A), \text{on}(X, a), \text{cl}(X), \text{on}(b, B), \text{cl}(b)$

The abstraction levels $\{1, 2, 3\}$ and $\{1, 4\}$ are logically equivalent. From a *minimum description length* (MDL) point of view, the latter is preferred. However, most humans will overlook that in (4) variable X can unify with b . But if we have to visually inspect 100 rules or 10, less rules are preferred. So, although logical representations may yield *comprehensible* abstractions, there are important trade-offs to be considered.

8.2.4 Algorithmic Aspects

Future research directions in the algorithmic aspects of relational RL should be mostly focused on constructive, incremental and more flexible representation induction algorithms. As we have discussed at the end of Chapter 3, representation drives control and vice versa. Therefore, representational devices should be structurally flexible to cope with the inherent concept drift in sequential decision making under uncertainty. So far, only the TGR and RUTREE algorithms introduce this kind of flexibility, but much more research is needed on efficient algorithms.

Another direction could be about *bottom-up* algorithms for structure learning, building generalized descriptions from individual, ground states. Most algorithms induce structures in a top-down fashion. However, much information about individual state transitions is lost in this way because only average statistical information is kept. Bottom-up generalization in RL is not used often, and new techniques have to be developed for this.

A last algorithmic future research direction is the use of powerful theorem provers to find exact solutions for RMDPs. In Chapter 6 we have seen several of such algorithms that rely on complex theorem proving. A number of approaches has moved to approximate representations to avoid such theorem proving. However, by making use of smarter representations, such as used in FODD, exact solutions might still be possible for even larger applications.

8.2.5 Theory

The development of theoretical insights has not kept pace with empirical work. A number of experimental approaches exists, each targeting a specific learning task or domain. However, development of theories on how and why some methods work is still developing. From a logical point of view, the deductive approaches are more amenable to formal analysis, but because the inductive approaches are based on the same semantics – an RMDP – both may benefit.

8.2.5.1 THEORETICAL FRAMEWORKS

Chapter 4 has defined the notions of an RMDPs and FORMs, based on standard FOL. Richer theories are needed to fully characterize the combination of FOL, probability and utility. Many of the model-based approaches can be seen as performing general, first-order reasoning, but much is unknown about their complexity. A starting point could be the analysis by Halpern (1990) on first-order probabilistic logic, which could be extended with a utility framework. Comparing syntactically different languages could provide insights in their relative complexity and efficiency when reasoning about decision-theoretic problems. What is also needed are comparisons between expressive formalisms such as situation calculus and less expressive ones such as probabilistic STRIPS, especially when they are used for the same RMDP.

On the semantic side, we have restricted most of our discussion to Herbrand interpretations for a given language. However, we can define much richer semantic structures, based on Tarskian or Kripke semantics, or extended with notions such as time as in temporal logics. Such richer semantic structures would require more general, logical languages extended with modal operators. Analysis of the resulting decision problems could map these into existing frameworks such as MDPs, POMDPs, SMDPs or more general frameworks. In the restricted setting of RMDPs, further analysis is needed on the notion of *families* of RMDPs. A value function is typically defined for a single RMDP, whereas several algorithms learn from a number of different members of an RMDP family. A theoretical characterization of how values learned on an abstraction level relate to the value functions of all members of this family is an open question. If the reward function depends on the number of objects in the domain, one could define *ranges* of values, which are related to the number of members of the family and their relative size. What the consequences are for learning algorithms should be investigated. For a fixed abstraction level, part of this question was approached by Guestrin *et al.* (2003a), but more analysis is needed. Characterizing what happens in infinite domains, both for learning and representing is an interesting topic. A theoretical investigation into the relation between propositional and first-order approaches in MDPs in line with the work done by De Raedt (1997, 1998) on concept learning would be highly useful.

Many different logical formalisms can be (and have been) employed for abstraction, such as Horn logic and description logics. It is important to provide *mappings* between these frameworks for comparison purposes. Also important are *representation theorems*: when inducing or defining an abstraction level, one has to consider the relation between an abstraction level and the underlying RMDP. For the deductive approaches (e.g. IDP), *soundness* of the reasoning process and simplification of expressions is important. For example, REBEL uses domain constraints to enforce this. Approaches based on fixed abstractions too have to ensure that the models they define, correspond to a correct underlying RMDP. For the inductive approaches, *completeness* is more an issue. Induced structures should be rich enough to model the underlying RMDP. A general issue to explore theoretically is the difference between value-based and policy-based approaches in the first-order setting, and more specifically their difference in computational complexity, especially sample complexity. Policy-based approaches have a much smaller search space, but it is more difficult to provide useful feedback to the learner.

8.2.5.2 CONVERGENCE AND OPTIMALITY

Convergence analysis is an important issue. Work on this is limited to some initial convergence results (see Ramon, 2005b,a; Kersting and De Raedt, 2004) and error bounds (see Guestrin *et al.*, 2003a; Sanner and Boutilier, 2005; Fern *et al.*, 2006) for specific systems. Relational abstraction is essentially a form of *function approximation*, for which – especially in model-free learning – convergence guarantees are scarce. Given the fact that most general relational abstractions can be viewed as *averaging* (Gordon, 1995c) useful bounds might be computed for fixed abstraction levels. However, because many inductive approaches employ PIAGET-3-learning, i.e. the abstraction level changes during the learning process, it is largely open how to approach convergence in these contexts (but see Ramon, 2005b,a). In general, one has to distinguish between *structural* convergence, i.e. obtaining a stable abstraction level and *value* convergence within an abstraction level. Not converging on the structural level does not have to exclude computing an optimal policy (see Chapter 6). The definition of *contraction mappings* with a structural dimension would be very interesting. The use of *action abstraction* is not considered in many existing convergence proofs in the propositional setting and is an important open problem.

8.2.5.3 COMPLEXITY

The trade-off between more powerful abstractions and the increased costs of manipulating relational structures should be investigated. For some classes of problems *propositionalization* might be more efficient in terms of computation, memory or sample complexity. It is interesting to study how the relation between abstraction levels and RMDPs changes if the latter grows, possibly to infinite size. Furthermore, using more expressive logics such as situation calculus enables more powerful abstraction (lower space complexity), but it comes with an increased cost of manipulating and learning them. Comparing propositional and first-order languages in terms of MDP-specific criteria, such as number of episodes, number of value backups and the relative sizes of value functions and policies are interesting and mostly absent. For the inductive approaches the number of needed samples is important (e.g. see Walsh and Littman, 2008, for a recent study), whereas in the deductive approaches the cost of manipulating expressions is an important issue.

Interestingly, the topic of complexity in relational RL (see also Wilson, 2004) can be approached from either the RL perspective or the logical perspective. From the viewpoint of RL, approximations can be introduced by making use of more efficient update schemes such as more efficient *exploration* and *asynchronous* versions of dynamic programming algorithms (such as variants of MPI). In this way, updates are concentrated on regions in the state(-action) space that are actually needed to derive optimal policies. Examples are *envelopes* (REBP), heuristic search (*approximate* FOVIA) and *guidance* in the Q-RRL system, but also efficient choices for domain instantiations such as used in the *random walks* approach in the LRW-API framework. From a logical viewpoint, approximations can be introduced by *coarser* abstraction levels. For example, the FOALP approach limits the representation of the value function to a finite set of *logical basis functions*. Related to this, the model-free approaches TG and RIB allow a small variance in the values of the abstraction level. Research on combinations of both types of approximations will prove to be vital to scale up to larger domains.

8.2.6 Agents, Cognitive Architectures, Reasoning and Transfer

Going beyond single RMDPs has been the topic of Chapter 7, where we have discussed the relationships between RMDPs, cognitive architectures, reasoning and transfer. In this general area there are many remaining challenges.

Concerning the single agent case, one direction is to enhance the many existing agent languages and architectures with the capability of decision-theoretic learning, for example as in the ICARUS system (Langley, 2006) or in SOAR (Nason and Laird, 2005). One of the main challenges is to *embed* the existing knowledge embodied by the approaches described in this book, into such architectures. Cognitive architectures have many reasoning capabilities that can be employed to reason over the experience gained in a learning process. This knowledge can be reused by forming plan libraries, or task hierarchies such that skills can be identified and transferred to other learning tasks. Transfer learning is an active topic in current ML, and for relational RL many challenges await, such as defining *what* exactly transfer is in such settings, *which* types of knowledge can be transferred, *how* it can be transferred, and what the benefits are. Because of the declarative nature of first-order logical knowledge bases, it also raises questions on how to use and transfer richly structured knowledge bases in learning.

Concerning the multi-agent system (MAS) case, an interesting direction is how to combine RMDPs with game-theoretic learning, such as reported by Finzi and Lukasiewicz (2004a) in the situation calculus. Combinations of multi-agent, modal logic approaches and game theory can provide a formal framework that could be used as a general starting point. The use of first-order representations in MAS contexts enables agents to reason about each other in terms of objects. The use of communication in a MAS would enable to transfer knowledge between agents, though here a challenge lies in the definition of shared languages and ontologies between agents in a MAS. The number of possible applications for relational RL in MAS are numerous (see Abul *et al.*, 2000; Stone and Veloso, 2000; Wiering *et al.*, 2000; Gu and Yang, 2004; Buçoni *et al.*, 2008, for examples).

8.2.7 Applications, Actions, Robots and Activities

Alike all other computational branches in AI, relational RL needs good applications to show the applicability and efficiency of its methods. The BLOCKS WORLD (Slaney and Thiébaux, 2001) is used in the majority of systems, but it should be viewed upon as – in analogy to genetics – a *Drosophila*. Among others, the following applications have also appeared in this book: TETRIS, DIGGER, TIC-TAC-TOE, various logistics and planning domains, BACKGAMMON, robot soccer, CHESS subgames, FREECRAFT, and dialogue systems. Although this represents a wide variety of problems, most of them are small and the purpose is to show the viability of the approaches. In order to more fully show the benefits of relational representations for MDPs, we need more, and more complex problems, preferably problems where propositional representations fail. Taking inspiration from the field of *probabilistic logic learning* (De Raedt and Kersting, 2003, 2004), *webmining*, *citation analysis* and *bio-informatics* applications are challenging domains due to their size and their intrinsically relational structure. First-order upgrades of RL *webspiders* (Rennie and McCallum, 1999) would be very interesting. Other domains in which learning and reasoning could be combined, are interesting, for example in *multi-agent* contexts. Here, many interesting problems can be explored, such as communication and cooperation (see

also Chapter 7). Furthermore, in the line of games such as CHESS and BACKGAMMON, it would be very interesting to approach the game of GO (see also Dabney and McGovern, 2007) or to find applications in the general area of *computer games* (Rabin, 2002; Amir and Doyle, 2002). Recent progress in real-time shooters such as *Unreal* show the viability of logic-based approaches (Jacobs *et al.*, 2005). A very interesting area consists of *real-time strategy* games, of which the domain taken by Guestrin *et al.* (2003a), – FREECRAFT – is an example. A more general area where relational RL could be applied is *human-computer interaction*, with applications in tutoring and training systems, virtual story telling and virtual worlds such as SECOND LIFE.

One area that will become more important and provides a more general test bed for relational RL and its extensions is what we call here *action and activity learning*. Much of what we have covered in this book is about *actions*. These can be simple as in STRIPS or more complex as in *hierarchical* decompositions, but also even more complex as in GOLOG programs or cognitive architectures. By moving to more complex actions, it becomes more appropriate to talk about *activities*; *dynamic*, *sequential* and *temporal* patterns of things that *happen*. Learning *relational* descriptions of behaviors and activities is an interesting and wide open research direction. Applications of dynamic behavior *recognition* and *prediction* can be found in many domains, for example in *video surveillance*, *robot control*, *social networks*, *physical movement tracking using sensors*, *shopping*, *(online) computer games* and many more (see Krueger *et al.*, 2007; Turaga *et al.*, 2008, for overviews).

Many types of new relational learning tasks can be defined, based on whether the behavior is *controlled* (as in RMDPs), or whether the behavior is monitored by an observer, whether the process is fully or partially observable (see Section 2.7), whether time plays an explicit role, or whether the sequential data is probabilistic or not. Some recent approaches target specific learning problems such as *online games* (e.g. Thon *et al.*, 2008), *location-based activity recognition* (Liao *et al.*, 2005), *action models of human daily activities* (Ortiz *et al.*, 2008), and *event detection on parking lots* (Tran and Davis, 2008). Two large fields that become more interested in relational representations and learning are *computer vision* and *robotics*. An interesting illustration of the use of relational representations in vision is described by Needham *et al.* (2005) who induce logical descriptions from video frames of people playing card games. These are then used to learn compact models of the dynamics of the game. As much of the work in robotics uses cameras to observe the world, many problems and opportunities are similar here. In the preceding chapters, we have already seen applications of RMDPs in manipulation tasks (Katz *et al.*, 2008) and action models that combine learning with low-level sensing with high-level planning (Kraft *et al.*, 2008; Petrick *et al.*, 2008). Especially in robotics, learning settings such as *imitation* and *behavioral cloning* (see Chapters 5 and 6) are useful. But also more novel techniques, such as *interactive perception* (Katz and Brock, 2008; Katz *et al.*, 2008), which is closely related to the RL approaches in this book.

In addition to new learning settings and new application domains, a very challenging problem in robotics and vision specifically is how to connect low-level observation data with high-level, relational descriptions. The *anchoring problem* (Coradeschi and Saffiotti, 2003) asks for a *grounding* of the high-level descriptions *in terms of* low-level observations. In this way, it is a more practical instance of the general *symbol grounding problem* (Harnad, 1990). Especially when the learner must generate its own representations of the world (PIAGET-3), these issues become more important. Additional pointers to the litera-

ture can be found in the special journal issue edited by Hertzberg and Saffiotti (2008) that deals with *semantic knowledge in robotics* and furthermore an investigation of (relational) machine learning techniques in robotics by Bratko (2008).

8.2.8 Benchmarks and Toolboxes

In order to compare various approaches, the field of relational RL needs *benchmark* problems. The international planning competition (Younes *et al.*, 2005) provides challenging problems, and several methods discussed in this book have entered the competition. We also need more standardized problems to compare different algorithms. For example, many papers use BLOCKS WORLDS but experimental setups, language biases and search heuristics, and the complexity of specific tasks, often differ much between approaches. In Chapter 5 we have described several methods that use TIC-TAC-TOE as their testing domain, but their results cannot be compared easily because their setups differ. This is also due to the use of different representational formalisms in each of the approaches. More work on standardized languages such as (P)PDDL and efficient translations to other languages, would be helpful for comparing systems in terms of efficiency, computational complexity and applicability. A last aspect that deserves attention, are publicly available systems and toolboxes. It is too soon for *off-the-shelf* algorithms for relational RL, given that these are not even available for the propositional setting. For supervised and unsupervised learning, many software programs exist such as MATLAB and WEKA, and it would be useful to have such software for (relational) RL in the future too.

8.2.9 Beyond the Markov Assumption

Algorithms for partially observable environments are an almost completely open research direction. Section 2.7 has described the basic POMDP setting, and nowadays many techniques have been developed (e.g. see Spaan, 2006, for pointers to the literature), both model-free and model-based. For the first-order setting, some initial research has begun. On the purely logical side, there are many connections with *modal* and *epistemic* logics, where belief states can consist of *sets* of states that the agent deems possible as being the "real" current state (see for references Fagin *et al.*, 1996; Wooldridge, 2002). Many action logics have been extended with epistemic notions (e.g. see Reiter, 2001, on the situation calculus). First-order versions of modal logic approaches introduce some new challenges concerning the interaction between modal operators and quantifiers. Probabilistic extensions of such logics have been studied for a long time (see Halpern, 2003, for an overview), and form the basis for the work in SRL (see Section 4.3.3). Upgrading POMDP representations and algorithms to the first-order case would enhance the applicability of relational RL towards more realistic problems, where there is some uncertainty about the agent's observations of the environment's state. Some have proposed representations of POMDPs (e.g. see Geffner and Bonet, 1998; Wang and Schmolze, 2005), but currently there are not many algorithms that compute solutions to such problems. Useful representations and algorithms could come from the field of SRL, and a first research direction consists of finding suitable representations to specify POMDPs. Most importantly, such representations should offer efficient reasoning and learning capabilities for use in efficient POMDP solution algorithms (see Croonenborghs *et al.*, 2006a, for related remarks).

Some model-free algorithms deal with non-Markovian problems (Itoh and Nakamura,

2004; Gretton, 2007a; Dabney and McGovern, 2007) or build partially-observable models (Shahaf and Amir, 2006). Finney *et al.* (2002b)'s approach yields partially observable problems because they use a deictic representation of an otherwise fully-observable, first-order state representation. Some initial work was done on predictive state representations (Wingate *et al.*, 2007) but also this area is still unexplored. The only model-based algorithm so far has been proposed by Wang (2007) who upgrades the incremental pruning algorithm to the first-order case. Many other types of such model-based POMDP algorithms could be obtained by using structured Bellman backup operators as defined in Chapter 6. An interesting approach for first-order domains would be to consider upgrading *point-based* approximation algorithms for POMDPs (Pineau *et al.*, 2006; Smith and Simmons, 2005; Spaan, 2006). Point-based algorithms use the same computational framework as exact algorithms, though backups (and computation of α -vectors) are only done for a small selected set of belief states. Given the huge spaces in first-order domains, as well as the considerable structure that is often present in those domains, this would be interesting.

8.3. Concluding Remarks

As we have argued throughout this book, first-order logical approaches offer many advantages for learning in richly structured, probabilistic domains. Such approaches are often highly comprehensible, and provide powerful abstractions over objects and relations. Many more developments are needed to further develop our understanding of these matters. It remains to be seen whether these "linguistically inspired" representations are the ultimate answer for computers. For humans, they are very meaningful, but it might well be that in the future, computational agents will develop their own representations of objects and their relations, grounded in their own sensory input (see also Sutton, 2007). Nevertheless, FOL representations have been shown to be vital in scaling up to larger and more structured problems and many challenges are available for further research.

The writing of this book has been a journey through a number of fields. For some of the topics that provide a foundation for relational RL, such as ILP, action formalisms and FOL, we have tried to cover the main concepts as good as possible, citing relevant works when needed, without having the pretence of providing a full reference to all these fields. For the (propositional) RL topics in Chapters 2 and 3 our guidelines for choosing the particular material were twofold. First, we aimed at presenting in significant detail all dimensions and approaches that provide the basis for relational RL, specifically for the Markov case. Second, for all five abstraction types identified in Chapter 3 we have tried to be as complete as possible when it comes to describing the *range* of possibilities for representations and algorithms in those types. For the field of relational RL itself, our goal has been to be *as complete as possible* in describing all approaches and citing all papers in the field. In this respect, this book can be seen as a full history of relational RL, ranging from its birth in 1998 with the work by Džeroski *et al.* (1998) until the end of the year 2008. In other words, the field of relational RL has reached its first decennium. We hope that this book, and much further research, will help it to grow and develop over the next decennium.

Bibliography

- Aberdeen, D. (2003), A (Revised) Survey of Approximative Methods for Solving Partially Observable Markov Decision Processes, unpublished manuscript, December 2003.
- Abul, O., Polat, F. and Alhajj, R. (2000), Multiagent Reinforcement Learning using Function Approximation, *IEEE Transactions on Systems, Man and Cybernetics, Part C: Application and Reviews*, volume 30(4), pp. 485–497.
- Agogino, A., Stanley, K. and Miikkulainen, R. (2000), Online Interactive Neuro-Evolution, *Neural Processing Letters*, volume 11(1), pp. 29–38.
- Agre, P. E. and Chapman, D. (1987), Pengi: An Implementation of a Theory of Activity, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 196–201.
- Aha, D., Kibler, D. and Albert, M. (1991), Instance-Based Learning Algorithms, *Machine Learning*, volume 6(1), pp. 37–66.
- Al-Ansari, M. A. and Williams, R. J. (1999), Efficient, Globally-Optimized Reinforcement Learning with the Parti-Game Algorithm, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 961–967.
- Aler, R., Borrajo, D. and Isasi, P. (2000), Knowledge Representation Issues in Control Knowledge Learning, in: Langley, P. (ed.), *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 1–8.
- Allender, E., Arora, S., Kearns, M., Moore, C. and Russell, A. (2002), A Note on the Representational Incompatibility of Function Approximation and Factored Dynamics, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 431–437.
- Alonso, E. and Mondragón, E. (2006), Associative Learning for Reinforcement Learning: Where Animal Learning and Machine Learning Meet, in: *Proceedings of the Fifth Symposium on Adaptive Agents and Multi-Agent Systems*, pp. 87–99.
- Alpaydin, E. (1991), GAL: Networks that Grow when they Learn and Shrink when they Forget, *Technical Report TR-91-032*, International Computer Science Institute, Berkeley.
- (2004), *Introduction to Machine Learning*, The MIT Press, Cambridge, Massachusetts.
- Amir, E. and Chang, A. (2008), Learning Partially Observable Deterministic Action Models, *Journal of Artificial Intelligence Research (JAIR)*, volume 33, pp. 349–402.
- Amir, E. and Doyle, P. (2002), Adventure Games: A Challenge for Cognitive Robotics, in: *Proceedings of the AAAI'02 Workshop on Cognitive Robotics*.
- Andersen, C. C. S. (2005), *Hierarchical Relational Reinforcement Learning*, Master's thesis, Aalborg University, Denmark.
- Anderson, C., Domingos, P. and Weld, D. S. (2002), Relational Markov Models and their Application to Adaptive Web Navigation, in: *Proceedings of the Eighth International Conference on Knowledge Discovery and Data Mining*, pp. 143–152.
- Anderson, C. W. (1993), Q-Learning with Hidden-Unit Restarting, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 81–88.
- (2000), Approximating a Policy can be easier than Approximating a Value Function, *Technical Report CS-00-101*, Computer Science Department, Colorado State University.
- Andre, D., Friedman, N. and Parr, R. (1998), Generalized Prioritized Sweeping, in: *Proceedings of the Neural*

BIBLIOGRAPHY

- Information Processing Conference (NIPS)*, pp. 1001–1007.
- Andre, D. and Russell, S. J. (2001), Programmable Reinforcement Learning Agents, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 1019–1025.
- (2002), State Abstraction for Programmable Reinforcement Learning Agents, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 119–125.
- Apt, K. and Bol, R. (1994), Logic Programming and Negation: a Survey, *Journal of Logic Programming*, volume 20, pp. 9–71.
- Arkin, R. C. (1998), *Behavior-Based Robotics*, The MIT Press, Cambridge, Massachusetts.
- Asadi, M. and Huber, M. (2005), Accelerating Action Dependent Hierarchical Reinforcement Learning through Autonomous Subgoal Discovery, in: Driessens, K., Fern, A. and van Otterlo, M. (eds.), *Proceedings of the ICML'05 Workshop on Rich Representations for Reinforcement Learning*, pp. 4–9.
- Asadi, M., Papudesi, V. and Huber, M. (2006), Learning Skill and Representation Hierarchies for Effective Control Knowledge Transfer, in: *ICML-06 Workshop on Structural Knowledge Transfer for Machine Learning*.
- Asgharbeygi, N., Nejati, N., Langley, P. and Arai, S. (2005), Guiding Inference in Reactive Agents through Relational Reinforcement Learning, in: *Proceedings of the International Conference on Inductive Logic Programming (ILP)*, pp. 20–37.
- Asgharbeygi, N., Stracuzzi, D. J. and Langley, P. (2006), Relational Temporal Difference Learning, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 49–56.
- Ash, T. (1989), Dynamic Node Creation in Backpropagation Networks, *Connection Science*, volume 1, pp. 365–375.
- Aycenina, M. (2002), Hierarchical Relational Reinforcement Learning, Stanford Doctoral Symposium, unpublished.
- Bacchus, F. (1990), LP, A Logic for Representing and Reasoning with Statistical Knowledge, *Computational Intelligence*, volume 6, pp. 209–231.
- Bacchus, F., Halpern, J. Y. and Levesque, H. J. (1999), Reasoning about Noisy Sensors and Effectors in the Situation Calculus, *Artificial Intelligence*, volume 111, pp. 171–208.
- Bäck, T. (1996), *Evolutionary Algorithms in Theory and Practice*, The Clarendon Press, New York.
- Bader, S. and Hitzler, P. (2005), Dimensions of Neural-Symbolic Integration – A Structured Survey, in: Artemov, S., Barringer, H., d'Avila Garcez, A. S., Lamb, L. C. and Woods, J. (eds.), *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 1, College Publications.
- Bader, S., Hitzler, P. and Hölldobler, S. (2006), The Integration of Connectionism and First-Order Knowledge Representation and Reasoning as a Challenge for Artificial Intelligence, *Journal of Information*, volume 9, pp. 7–20.
- Baillargeon, R. (1999), The Object Concept Revisited: New Directions in the Investigation of Infant's Physical Knowledge, in: Margolis (1999), chapter 25, pp. 571–612.
- Bain, M. and Sammut, C. (1995), A Framework for Behavioral Cloning, in: Muggleton, S. H., Furakawa, K. and Michie, D. (eds.), *Machine Intelligence 15*, Oxford University Press, pp. 103–129.
- Baird, L. C. (1995), Residual Algorithms: Reinforcement Learning with Function Approximation, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 30–37.
- (1999), *Reinforcement Learning through Gradient Descent*, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh.
- Baird, L. C. and Moore, A. W. (1999), Gradient Descent for General Reinforcement Learning, in: *Proceedings of the Neural Information Processing Conference (NIPS)*.
- Bakker, B. (2004), *The State of Mind: Reinforcement Learning with Recurrent Neural Networks*, Ph.D. thesis, University of Leiden, The Netherlands, Faculteit Wiskunde, Natuurwetenschappen en Geneeskunde.
- Bakker, B. and Schmidhuber, J. H. (2004), Hierarchical Reinforcement Learning Based on Subgoal Discovery and Subpolicy Specialization, in: *Proceedings of the 8-th Conference on Intelligent Autonomous Systems, IAS-8, Amsterdam, The Netherlands*, pp. 438–445.
- Bakker, B. and van der Voort van der Kleij, G. (2000), Trading off Perception with Internal State: Reinforcement Learning and analysis of Q-Elman Networks in a Markovian Task, in: *Proceedings of the International Joint Conference on Neural Networks 2000*, volume III, pp. 213–218.
- Baral, C. and Tran, S. C. (1998), Relating Theories of Actions and Reactive Control, *Electronic Transactions*

- on *Artificial Intelligence*, volume 2, pp. 211–271.
- Bartlett, P. L. (2003), An Introduction to Reinforcement Learning Theory: Value Function Methods, in: *Advanced Lectures on Machine Learning*, volume 1600 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 184–202.
- Barto, A. G., Bradtke, S. J. and Singh, S. (1995), Learning to Act Using Real-Time Dynamic Programming, *Artificial Intelligence*, volume 72(1), pp. 81–138.
- Barto, A. G. and Dietterich, T. G. (2004), Reinforcement Learning and its Relationship to Supervised Learning, in: Si, J., Barto, A. G., Powell, W. B. and Wunsch II, D. (eds.), *Handbook of Learning and Approximate Dynamic Programming*, Wiley Interscience/IEEE Press, Piscataway, NJ, pp. 47–64.
- Barto, A. G. and Mahadevan, S. (2003), Recent Advances in Hierarchical Reinforcement Learning, *Discrete event systems*, volume 13(4), pp. 341–379.
- Barto, A. G., Sutton, R. S. and Anderson, C. W. (1983), Neuronlike Elements that can Solve Difficult Learning Control Problems, *IEEE Transactions on Systems, Man, and Cybernetics*, volume 13, pp. 835–846.
- Baum, E. B. (1998), Manifesto for an Evolutionary Economics of Intelligence, in: Bishop, C. M. (ed.), *Neural Networks and Machine Learning*, volume 168 of *NATO ASI Series F*, Springer-Verlag.
- (1999), Toward a Model of Intelligence as an Economy of Agents, *Machine Learning*, volume 35(2), pp. 155–185.
- (2004), *What is Thought?*, The MIT Press, Cambridge, Massachusetts.
- Baum, J. and Nicholson, A. (1998), Dynamic Non-Uniform Abstractions for Approximate Planning in Large Structured Stochastic Domains, in: *In Proceedings of the 5th Pacific Rim International Conference on Artificial Intelligence*, pp. 587–598.
- Bazen, A. M. and Gerez, S. H. (2000), Directional Field Computation for Fingerprints Based on the Principal Component Analysis of Local Gradients, in: *Proceedings of ProRISC2000, 11th Annual Workshop on Circuits, Systems and Signal Processing*.
- Bazen, A. M., van Otterlo, M., Gerez, S. H. and Poel, M. (2001), A Reinforcement Learning Agent for Minutiae Extraction from Fingerprints, in: *Proceedings of the Belgium-Netherlands Artificial Intelligence Conference (BNAIC'01)*, pp. 329–336.
- Bellemare, M. G., Precup, D. and Rivest, F. (2004), Reinforcement Learning using Cascade-Correlation Networks, *Technical Report RL-3.04*, McGill University, Canada.
- Bellman, R. E. (1957), *Dynamic Programming*, Princeton University Press, Princeton, New Jersey.
- Benson, S. S. (1996), *Learning Action Models for Reactive Autonomous Agents*, Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, California.
- Bergadano, F. and Gunetti, D. (1995), *Inductive Logic Programming: From Machine Learning to Software Engineering*, The MIT Press, Cambridge, Massachusetts.
- Bertsekas, D. P. (1995), *Dynamic Programming and Optimal Control, volumes 1 and 2*, Athena Scientific, Belmont, MA.
- Bertsekas, D. P. and Castañón, D. A. (1989), Adaptive Aggregation Methods for Infinite Horizon Dynamic Programming, *IEEE Transactions on Automatic Control*, volume 34(6), pp. 589–598.
- Bertsekas, D. P. and Tsitsiklis, J. (1996), *Neuro-Dynamic Programming*, Athena Scientific, Belmont, MA.
- Billard, A. and Hayes, G. (1999), DRAMA, a Connectionist Architecture for Control and Learning in Autonomous Robots, *Adaptive Behavior Journal*, volume 7(1).
- Bishop, C. M. (1995), *Neural Networks for Pattern Recognition*, Clarendon Press, Oxford.
- Blockeel, H. (1998), *Top-Down Induction of First Order Logical Decision Trees*, Ph.D. thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium.
- Blockeel, H. and De Raedt, L. (1998), Top-Down Induction of First-Order Logical Decision Trees, *Artificial Intelligence*, volume 101(1–2), pp. 285–297.
- Blockeel, H., De Raedt, L., Jacobs, N. and Dempoen, B. (2000a), Scaling Up Inductive Logic Programming by Learning from Interpretations, *Technical Report CW-297*, Department of Computer Science, Catholic University of Leuven, Belgium.
- Blockeel, H., De Raedt, L. and Ramon, J. (1998), Top-Down Induction of Clustering Trees, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 55–63.
- Blockeel, H., Dehaspe, L., Dempoen, B., Janssens, G., Ramon, J. and Vandecasteele, H. (2000b), Executing query packs in ILP, in: Cussens, J. and Frisch, A. (eds.), *Proceedings of the International Conference on*

BIBLIOGRAPHY

- Inductive Logic Programming (ILP)*, volume 1866 of *Lecture Notes in Artificial Intelligence*, Springer, pp. 60–77.
- Blum, A. and Langley, P. (1997), Selection of Relevant Features and Examples in Machine Learning, *Artificial Intelligence*, volume 97, pp. 245–271.
- Blythe, J. (1999), An Overview of Planning Under Uncertainty, *Lecture Notes in Computer Science*, volume 1600, pp. 85–110.
- Bonet, B. and Geffner, H. (2003a), Faster Heuristic Search Algorithms for Planning with Uncertainty and Full Feedback, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1233–1238.
- (2003b), Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming, in: *Proceedings of the International Conference on Artificial Intelligence Planning Systems (ICAPS)*, pp. 12–21.
- Bongard, M. (1970), *Pattern Recognition*, Hayden Book Company.
- Booker, L. B., Goldberg, D. E. and Holland, J. H. (1989), Classifier Systems and Genetic Algorithms, in: *Artificial Intelligence*, volume 40, pp. 235–282.
- Botta, M., Giordana, A. and Piola, R. (1997), FONN: Combining First-Order Logic with Connectionist Learning, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 48–56.
- Boutilier, C. (1997), Correlated Action Effects in Decision-Theoretic Regression, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 30–37.
- (1999), Knowledge Representation for Stochastic Decision Processes, *Lecture Notes in Computer Science*, volume 1600, pp. 111–152.
- (2001), Planning and Programming with First-order Representations of Markov Decision Processes, accompanying paper for the invited talk at TARK VIII.
- Boutilier, C., Dean, T. and Hanks, S. (1999), Decision Theoretic Planning: Structural Assumptions and Computational Leverage, *Journal of Artificial Intelligence Research*, volume 11, pp. 1–94.
- Boutilier, C. and Dearden, R. W. (1994), Using Abstractions for Decision-Theoretic Planning with Time Constraints, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 1016–1022.
- (1996), Approximating Value Trees in Structured Dynamic Programming, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 54–62.
- Boutilier, C., Dearden, R. W. and Goldszmidt, M. (1995), Exploiting Structure in Policy Construction, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1104–1111.
- (2000a), Stochastic Dynamic Programming with Factored Representations, *Artificial Intelligence*, volume 121(1–2), pp. 49–107.
- Boutilier, C. and Poole, D. (1996), Computing Optimal Policies for Partially Observable Decision Processes using Compact Representations, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 1168–1175.
- Boutilier, C., Reiter, R. and Price, B. (2001), Symbolic Dynamic Programming for First-Order MDP's, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 690–697.
- Boutilier, C., Reiter, R., Soutchanski, M. and Thrun, S. (2000b), Decision-Theoretic, High-Level Agent Programming in the Situation Calculus, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 355–362.
- Boyan, J. A. and Littman, M. L. (2000), Exact Solutions to Time-Dependent MDPs, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 1026–1032.
- Boyan, J. A. and Moore, A. W. (1995), Generalization in Reinforcement Learning: Safely Approximating the Value Function, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 369–376.
- Brachman, R. J. and Levesque, H. J. (2004), *Knowledge Representation and Reasoning*, Morgan Kaufmann Publishers, San Francisco, CA.
- Brafman, R. I. (2008), Relational Preference Rules for Control, in: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*.
- Brafman, R. I. and Tennenholtz, M. (2002), R-MAX - A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning, *Journal of Machine Learning Research (JMLR)*, volume 3, pp. 213–231.
- Braitenberg, V. (1984), *Vehicles: Experiments in Synthetic Psychology*, The MIT Press, Cambridge, Massachusetts.
- Bratko, I. (2001), *Prolog, Programming for Artificial Intelligence*, Addison-Wesley, 3rd edition.

- (2008), An Assessment of Machine Learning Methods for Robotic Discovery, in: *Proceedings of the International Conference on Information Technology Interfaces (ITI)*, pp. 53–60.
- Breiman, L., Friedman, J. H., Olshen, R. A. and Stone, C. J. (1984), *Classification and Regression Trees*, The Wadsworth Statistics/Probability Series, Wadsworth and Brooks/Cole Advanced Books and Software.
- Broersen, J., Dastani, M., Hulstijn, J. and van der Torre, L. (2002), Goal Generation in the BOID Architecture, *Cognitive Science Quarterly*, volume 2(3–4), pp. 428–447.
- Brooks, R. A. (1991), Intelligence without Representation, *Artificial Intelligence*, volume 47, pp. 139–159.
- Browne, A. and Sun, R. (2001), Connectionist Inference Models, *Neural Networks*, volume 14(10), pp. 1331–1355.
- Bruske, J., Ahrns, I. and Sommer, G. (1997), An Integrated Architecture for Learning of Reactive Behaviors based on Dynamic Cell Structures, *Robotics and Autonomous Systems*, volume 22, pp. 87–101.
- Buçoni, L., Babuška, R. and de Schutter, B. (2008), A Comprehensive Survey of Multiagent Reinforcement Learning, *IEEE Transactions on Systems, Man, and Cybernetics – Part C: Applications and Reviews*, volume 38(2).
- Buntine, W. (1988), Generalized Subsumption and its Applications to Induction and Redundancy, *Artificial Intelligence*, volume 36(2), pp. 149–176.
- Caruana, R. (1997), Multi-Task Learning, *Machine Learning*, volume 28(1), pp. 41–75.
- Caruana, R., Lawrence, S. and Giles, C. L. (2001), Overfitting in Neural Networks: Backpropagation, Conjugate Gradient, and Early Stopping, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 402–408.
- Castilho, M., Kunzle, L. A., Lecheta, E., Palodeto, V. and Silva, F. (2004), An Investigation on Genetic Algorithms for Generic STRIPS Planning, in: *IBERAMIA 2004*, volume 3315 of *Lecture Notes in Computer Science*, pp. 185–194.
- Chapman, D. and Kaelbling, L. P. (1991), Input Generalization in Delayed Reinforcement Learning: An algorithm And Performance Comparisons, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 726–731.
- Choi, D., Kaufman, M., Langley, P., Nejati, N. and Shapiro, D. (2004), An Architecture for Persistent Reactive Behavior, in: *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi Agent Systems*, pp. 988–995.
- Claplin, H. (ed.) (2002), *Philosophy of Mental Representation*, Oxford University Press, Oxford, UK.
- Clark, A. (1997), *Being There*, The MIT Press, Cambridge, Massachusetts.
- Clark, A. and Thornton, C. (1997), Trading Spaces: Computation, Representation, and the Limits of Uninformed Learning, *Behavioral and Brain Sciences*, volume 20(506), pp. 57–90.
- Cloete, I. and Zurada, J. M. (2000), *Knowledge-Based Neurocomputing*, The MIT Press, Cambridge, Massachusetts.
- Cocora, A., Kersting, K., Plagemann, C., Burgard, W. and De Raedt, L. (2006), Learning Relational Navigation Policies, in: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.
- Cole, J., Lloyd, J. W. and Ng, K. S. (2003), Symbolic Learning for Adaptive Agents, in: *Proceedings of the Annual Partner Conference, Smart Internet Technology Cooperative Research Centre*, http://csl.anu.edu.au/jwl/crc_paper.pdf.
- Coradeschi, S. and Saffiotti, A. (2003), An Introduction to the Anchoring Problem, *Robotics and Autonomous Systems*, volume 43(2–3), pp. 85–96.
- Crites, R. H. and Barto, A. G. (1996), Improving Elevator Performance Using Reinforcement Learning, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 1017–1023.
- Croonenborghs, T., Driessens, K. and Bruynooghe, M. (2007a), Learning Relational Options for Inductive Transfer in Relational Reinforcement Learning, in: *Proceedings of the International Conference on Inductive Logic Programming (ILP)*.
- Croonenborghs, T., Ramon, J., Blockeel, H. and Bruynooghe, M. (2006a), Model-assisted Approaches for Relational Reinforcement Learning: Some Challenges for the SRL Community, in: *Proceedings of the ICML-06 Workshop on Open Problems in Statistical Relational Learning*.
- (2007b), Online Learning and Exploiting Relational Models in Reinforcement Learning, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 726–731.

BIBLIOGRAPHY

- Croonenborghs, T., Ramon, J. and Bruynooghe, M. (2004), Towards Informed Reinforcement Learning, in: *Proceedings of the Workshop on Relational Reinforcement Learning at ICML'04*.
- Croonenborghs, T., Tuyls, K., Ramon, J. and Bruynooghe, M. (2006b), Multi-Agent Relational Reinforcement Learning: Explorations in Multi-State Coordination Tasks, in: *Proceedings of LAMAS 2005*, pp. 192–206.
- Cumby, C. and Roth, D. (2003), On Kernel Methods for Relational Learning, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 107–114.
- Dabney, W. and McGovern, A. (2006), The Thing We tried that Worked: Utile Distinctions for Relational Reinforcement Learning, in: *Proceedings of the ICML-06 Workshop on Open Problems in Statistical Relational Learning*.
- (2007), Utile Distinctions for Relational Reinforcement Learning, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 738–743.
- Dai, P. and Goldsmith, J. (2007), Topological Value Iteration Algorithm for Markov Decision Processes, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1860–1865.
- Damasio, A. R. (1994), *Descartes' Error: Emotion, Reason and the Human Brain*, Gosset/Putnam, New York, NY.
- Dastani, M., de Boer, F. S., Dignum, F. and Meyer, J. J. (2003a), Programming Agent Deliberation: An Approach Illustrated Using the 3APL Language, in: *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pp. 97–104.
- Dastani, M., Dignum, F. and Meyer, J. J. (2003b), Autonomy and Agent Deliberation, in: *Proceedings of The First International Workshop on Computational Autonomy - Potential, Risks, Solutions (Autonomous 2003)*.
- Dastani, M. and Meyer, J. J. (2006), Programming Agent with Emotions, in: *Proceedings of the European Conference on Artificial Intelligence (ECAI)*.
- Dastani, M., van Riemsdijk, B., Dignum, F. and Meyer, J.-J. (2003c), A Programming Language for Cognitive Agents: Goal-directed 3APL, in: *Proceedings the ProMAS 2003 Workshop at AAMAS'03*.
- d'Avila Garcez, A. S., Broda, K. and Gabbay, D. M. (2001), Symbolic Knowledge Extraction from Trained Neural Networks: A Sound Approach, *Artificial Intelligence*, volume 125, pp. 155–207.
- Davis, M. (2000), *Engines of Logic: Mathematicians and The Origin of The Computer*, Norton & Company Inc., New York.
- Dayan, P. (1998), Feudal Q-Learning, unpublished Technical Report.
- (2000), Reinforcement Learning, in: Gallistel, C. R. (ed.), *Steven's Handbook of Experimental Psychology*, Wiley, New York.
- Dayan, P. and Abbott, L. F. (2001), Classical Conditioning and Reinforcement Learning, in: *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*, chapter 9, The MIT Press, Cambridge, Massachusetts.
- Dayan, P. and Hinton, G. E. (1993), Feudal Reinforcement Learning, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 271–278.
- de Jong, E. D. (2000), *Autonomous Formation of Concepts and Communication*, Ph.D. thesis, Vrije Universiteit Brussel.
- de la Rosa, T., Jimenez, S. and Borrajo, D. (2008), Learning Relational Decision Trees for Guiding Heuristic Planning, in: *Proceedings of the International Conference on Artificial Intelligence Planning Systems (ICAPS)*.
- De Raedt, L. (1997), Logical Settings for Concept-Learning, *Artificial Intelligence*, volume 95(1), pp. 187–201.
- (1998), Attribute Value Learning versus Inductive Logic Programming: The Missing Links (Extended Abstract), in: *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 1–8.
- (2008), *Logical and Relational Learning*, Springer.
- De Raedt, L., Blockeel, H., DeHaspe, L. and Van Laer, W. (2001), Three Companions for Data Mining in First Order Logic, in: Džeroski and Lavrac (2001b), chapter 5, pp. 105–139.
- De Raedt, L. and Dehaspe, L. (1997), Clausal Discovery, *Machine Learning*, volume 26, pp. 99–146.
- De Raedt, L. and Kersting, K. (2003), Probabilistic Logic Learning, *ACM-SIGKDD Explorations, special issue on Multi-Relational Data Mining*, volume 5(1), pp. 31–48.

BIBLIOGRAPHY

- (2004), Probabilistic Inductive Logic Programming, in: *Proceedings of the International Conference on Algorithmic Learning Theory (ALT)*, pp. 19–36.
- De Raedt, L., Kimmig, A. and Toivonen, H. (2007a), Probabilistic Explanation Based Learning, in: *Proceedings of the European Conference on Machine Learning (ECML)*, pp. 176–187.
- (2007b), ProbLog: A Probabilistic Prolog and Its Application in Link Discovery, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 2462–2467.
- de Salvo Braz, R., Amir, E. and Roth, D. (2005), Lifted First-Order Probabilistic Inference, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1319–1325.
- Dean, T. and Givan, R. (1997), Model Minimization in Markov Decision Processes, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 106–111.
- Dean, T., Givan, R. and Kim, K. E. (1998), Solving Stochastic Planning Problems with Large State and Action Spaces, in: *Proceedings of the International Conference on Artificial Intelligence Planning Systems (ICAPS)*, pp. 102–110.
- Dean, T., Givan, R. and Leach, S. (1997), Model Reduction Techniques for Computing Approximately Optimal Solutions for Markov Decision Processes, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 124–131.
- Dean, T., Kaelbling, L. P., Kirman, J. and Nicholson, A. (1995), Planning under Time Constraints in Stochastic Domains, *Artificial Intelligence*, volume 76, pp. 35–74.
- Dean, T. and Kanawaza, K. (1989), A Model for Reasoning about Persistence and Causation, *Computational Intelligence*, volume 5, pp. 142–150.
- Dean, T. and Lin, S. H. (1995), Decomposition Techniques for Planning in Stochastic Domains, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1121–1129.
- Dearden, R. W. (2000), *Learning and Planning in Structured Worlds*, Ph.D. thesis, Department of Computer Science, The University of British Columbia.
- (2001), Structured Prioritized Sweeping, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 82–89.
- Dearden, R. W. and Boutilier, C. (1997), Abstraction and Approximate Decision-Theoretic Planning, *Artificial Intelligence*, volume 89(1–2), pp. 219–283.
- Degrís, T., Sigaud, O. and Wuillemin, P. H. (2006), Learning the Structure of Factored Markov Decision Processes in Reinforcement Learning Problems, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 257–264.
- Dempster, A. P., Laird, N. M. and Rubin, D. B. (1977), Maximum Likelihood from Incomplete Data via the EM algorithm, *Journal of the Royal Statistical Society*, volume 39(1), pp. 1–38.
- Denecker, M., Martens, B. and De Raedt, L. (1996), On the Difference between Abduction and Induction: A Model-Theoretic Perspective, in: *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pp. 19–22.
- Dennett, D. C. (1987), *The Intentional Stance*, The MIT Press, Cambridge, Massachusetts.
- (1998), *Brainchildren: Essays on Designing Minds*, Penguin Books, Great Britain.
- Dietterich, T. G. (1998), The MAXQ Method for Hierarchical Reinforcement Learning, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 118–125.
- (2000a), Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition, *Journal of Artificial Intelligence Research (JAIR)*, volume 13, pp. 227–303.
- (2000b), An Overview of MAXQ Hierarchical Reinforcement Learning, in: *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA)*, Lecture Notes in Artificial Intelligence, Springer Verlag, New York, pp. 26–44.
- (2003), Learning and Reasoning, unpublished.
- Dietterich, T. G. and Flann, N. S. (1995), Explanation-Based Learning and Reinforcement Learning: A Unified Approach, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 176–184.
- (1997), Explanation-Based Learning and Reinforcement Learning: A Unified View, *Machine Learning*, volume 28(503), pp. 169–210.
- Dietterich, T. G., Getoor, L. and Murphy, K. (eds.) (2004), *Proceedings of the ICML-2004 Workshop on Statistical Relational Learning and its Connections to Other Fields (SRL'04)*.
- Dietterich, T. G. and Wang, X. (2002), Batch Value Function Approximation using Support Vectors, in:

BIBLIOGRAPHY

- Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 1491–1498.
- Digney, B. (1998), Learning Hierarchical Reinforcement Learning for Multiple Tasks and Changing Environments, in: *From Animals to Animats: Proceedings of The International Conference on Simulation of Adaptive Behavior (SAB)*, pp. 321–330.
- Dijkstra, E. W. (1975), Guarded Commands, Nondeterminacy and Formal Derivation of Programs, *Communications of the ACM*, volume 18(8), pp. 453–457.
- d’Inverno, M., Kinny, D., Luck, M. and Wooldridge, M. (1997), A Formal Specification of dMARS, in: *Agent Theories, Architectures, and Languages*, pp. 155–176.
- Diuk, C., Cohen, A. and Littman, M. L. (2008), An Object-Oriented Representation for Efficient Reinforcement Learning, in: *Proceedings of the International Conference on Machine Learning (ICML)*.
- Divina, F. (2004), *Hybrid Genetic Relational Search for Inductive Learning*, Ph.D. thesis, Department of Computer Science, Vrije Universiteit, Amsterdam, the Netherlands.
- (2006), Evolutionary Concept Learning in First Order Logic, *AI Communications*, volume 19(1), pp. 13–33.
- Dixon, K. R., Malak, M. J. and Khosla, P. K. (2000), Incorporating Prior Knowledge and Previously Learned Information into Reinforcement Learning Agents, *Technical report*, Institute for Complex Engineered Systems, Carnegie Mellon University.
- Doan, A. and Haddawy, P. (1995), Abstraction for Decision-Theoretic Planning, unpublished.
- Domingos, P. (2008), What’s Missing in AI: The Interface Layer, in: Cohen, P. (ed.), *Artificial Intelligence: The First Hundred Years*, AAAI Press, Menlo Park, CA.
- Domingos, P. and Richardson, M. (2004), Markov Logic: A Unifying Framework for Statistical Relational Learning, in: *Proceedings of the ICML-2004 Workshop on Statistical Relational Learning and its Connections to Other Fields*, pp. 49–54.
- Dorigo, M. and Bersini, H. (1994), A Comparison of Q-Learning and Classifier Systems, in: *From Animals to Animats: Proceedings of The International Conference on Simulation of Adaptive Behavior (SAB)*, pp. 248–255.
- Dorigo, M. and Colombetti, M. (1997), *Robot Shaping: An Experiment in Behavior Engineering*, The MIT Press, Cambridge, Massachusetts.
- Drescher, G. (1991), *Made-Up Minds: A Constructivist Approach to Artificial Intelligence*, The MIT Press, Cambridge, Massachusetts.
- Driessens, K. (2001a), Relational Reinforcement Learning, in: *Multi-Agent Systems and Applications*, Springer-Verlag, pp. 271–280.
- (2001b), Relational Reinforcement Learning, advanced Course on Artificial Intelligence : The Third European Agent Systems Summer School, Prague, Czech Republik, July 2–13, 2001.
- (2004), *Relational Reinforcement Learning*, Ph.D. thesis, Department of Computer Science, Catholic University of Leuven, Belgium.
- (2005), Relational Reinforcement Learning (Thesis Description), *AI Communications*, volume 18(1).
- Driessens, K. and Blockeel, H. (2001), Learning Digger using Hierarchical Reinforcement Learning for Concurrent Goals, in: *Proceedings of the European Workshop on Reinforcement Learning (EWRL)*.
- Driessens, K. and Džeroski, S. (2002a), Integrating Experimentation and Guidance in Relational Reinforcement Learning, in: *Proceedings of the Nineteenth International Conference on Machine Learning*, pp. 115–122.
- (2002b), On Using Guidance in Relational Reinforcement Learning, in: *Proceedings of Twelfth Belgian-Dutch Conference on Machine Learning*, pp. 31–38.
- (2004), Integrating Experimentation and Guidance in Relational Reinforcement Learning, *Machine Learning*, volume 57, pp. 271–304.
- (2005), Combining Model-Based and Instance-Based Learning for First Order Regression, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 193–200.
- Driessens, K. and Ramon, J. (2003), Relational Instance Based Regression for Relational Reinforcement Learning, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 123–130.
- Driessens, K., Ramon, J. and Blockeel, H. (2001a), Speeding Up Relational Reinforcement Learning through the Use of an Incremental First Order Decision Tree Algorithm, in: *Proceedings of ECML - European Conference on Machine Learning*, volume 2167 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag,

- pp. 97–108.
- (2001b), Speeding Up Relational Reinforcement Learning through the Use of an Incremental First Order Decision Tree Algorithm, in: *Proceedings of the Belgium-Netherlands Artificial Intelligence Conference (BNAIC)*.
- Driessens, K., Ramon, J. and Croonenborghs, T. (2006a), Transfer Learning for Reinforcement Learning through Goal and Policy Parameterization, in: *Proceedings of the ICML'06 Workshop on Structural Knowledge Transfer for Machine Learning*.
- Driessens, K., Ramon, J. and Gärtner, T. (2006b), Graph Kernels and Gaussian Processes for Relational Reinforcement Learning, *Machine Learning*, volume 64(1–3), pp. 91–119.
- Džeroski, S. (2001), Relational Data Mining Applications: An Overview, in: Džeroski and Lavrac (2001b), chapter 14, pp. 339–364.
- (2002), Relational Reinforcement Learning for Agents in Worlds with Objects, in: *Proceedings of the Symposium on Adaptive Agents and Multi-Agent Systems (AISB'02)*, pp. 1–8.
- (2003), Learning in Rich Representations: Inductive Logic Programming and Computational Scientific Discovery, in: *Proceedings of the International Conference on Inductive Logic Programming (ILP)*, pp. 346–349.
- Džeroski, S., De Raedt, L. and Blockeel, H. (1998), Relational Reinforcement Learning, in: Shavlik, J. (ed.), *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 136–143.
- Džeroski, S., De Raedt, L. and Driessens, K. (2001a), Relational Reinforcement Learning, *Machine Learning*, volume 43, pp. 7–52.
- (2001b), Relational Reinforcement Learning, *Technical Report CW-311*, Department of Computer Science, Catholic University of Leuven.
- Džeroski, S. and Lavrac, N. (2001a), An Introduction to Inductive Logic Programming, in: Džeroski and Lavrac (2001b), chapter 3, pp. 48–73.
- Džeroski, S. and Lavrac, N. (eds.) (2001b), *Relational Data Mining*, Springer-Verlag, Berlin.
- Edelkamp, S. and Hoffman, J. (2004), PDDL 2.2: The Language for the Classical Part of the 4th International Planning Competition, *Technical Report Technical Report 195*, Albert-Ludwigs Universitaet, Freiburg, Germany.
- Ellman, T. (1989), Explanation-Based Learning: A Survey of Programs and Perspectives, *ACM Computing Surveys*, volume 21(2).
- Elman, J. L. (1990), Finding Structure in Time, *Cognitive Science*, volume 14(2), pp. 179–211.
- Elman, J. L., Bates, E. A., Johnson, M. H., Karmiloff-Smith, A., Parisi, D. and Plunkett, K. (1996), *Rethinking Innateness: A Connectionist Perspective on Development*, The MIT Press, Cambridge, Massachusetts.
- Engel, Y., Mannor, S. and Meir, R. (2005), Reinforcement Learning with Kernels and Gaussian Processes, in: Driessens, K., Fern, A. and van Otterlo, M. (eds.), *Proceedings of the ICML'05 Workshop on Rich Representations for Reinforcement Learning*, pp. 16–20.
- Erdmann, M. (1986), Using Backprojections for Fine Motion Planning with Uncertainty, *International Journal of Robotics Research*, volume 5(1), pp. 19–45.
- Ernst, D., Geurts, P. and Wehenkel, L. (2005), Tree-Based Batch Mode Reinforcement Learning, *Journal of Machine Learning Research*, volume 6, pp. 503–556.
- Erol, K., Hendler, J. and Nau, D. S. (1994), HTN Planning: Complexity and Expressivity, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 1123–1128.
- Even-Dar, E. and Mansour, Y. (2003), Approximate Equivalence of Markov Decision Processes, in: *Proceedings of the International Conference on Computational Learning Theory (COLT)*, volume 2777 of *Lecture Notes in Artificial Intelligence*, pp. 581–594.
- Fagin, R., Halpern, J. Y., Moses, Y. and Vardi, M. Y. (1996), *Reasoning about Knowledge*, The MIT Press, Cambridge, Massachusetts.
- Fahlman, S. E. and Lebiere, C. L. (1990), The Cascade-Correlation Learning Architecture, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, p. 524.
- Feng, Z. (2004), Towards better Scalability in Solving MDPs and POMDPs (Extended Abstract), in: *ICAPS 2004 Doctoral Consortium*.
- Feng, Z., Dearden, R. W., Meuleau, N. and Washington, R. (2004), Dynamic Programming for Structured Continuous Markov Decision Problems, in: *Proceedings of the Conference on Uncertainty in Artificial*

BIBLIOGRAPHY

- Intelligence (UAI)*, pp. 154–161.
- Feng, Z. and Hansen, E. A. (2002), Symbolic LAO* Search for Factored Markov Decision Processes, in: *AIPS 2002 Workshop on Planning via Model Checking*.
- Feng, Z., Hansen, E. A. and Zilberstein, S. (2003), Symbolic Generalization for On-Line Planning, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 209–216.
- Ferber, J. (1999), *Multi-Agent Systems*, Pearson Education Unlimited, Great Britain.
- Ferguson, D. and Stentz, A. (2004), Focussed Dynamic Programming: Extensive Comparative Results, *Technical Report CMU-RI-TR-04-13*, Robotics Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.
- Fern, A., Yoon, S. W. and Givan, R. (2003), Approximate Policy Iteration with a Policy Language Bias, in: *Proceedings of the Neural Information Processing Conference (NIPS)*.
- (2004a), Learning Domain-Specific Control Knowledge from Random Walks, in: *Proceedings of the International Conference on Artificial Intelligence Planning Systems (ICAPS)*, pp. 191–199.
- (2004b), Relational Reinforcement Learning for Classical Planning (Extended Abstract), in: *Proceedings of the Workshop on Relational Reinforcement Learning at ICML'04*.
- (2006), Approximate Policy Iteration with a Policy Language Bias: Solving Relational Markov Decision Processes, *Journal of Artificial Intelligence Research (JAIR)*, volume 25, pp. 75–118, special issue on the International Planning Competition 2004.
- (2007), Reinforcement Learning in Relational Domains: A Policy-Language Approach, in: Getoor and Taskar (2007).
- Ferns, N., Panangaden, P. and Precup, D. (2004), Metrics for Finite Markov Decision Processes, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 162–169.
- Ferrein, A., Fritz, C. and Lakemeyer, G. (2004), On-Line Decision-Theoretic Golog for Unpredictable Domains, in: *Proceedings of the 27th German Conference on AI (KI'04)*.
- Fiesler, E. and Cios, K. (1997), Supervised Ontogenic Networks, in: Fiesler, E. and Beale, R. (eds.), *Handbook of Neural Computation*, chapter C1.7, Institute of Physics and Oxford University Press, New York, New York.
- Fikes, R. E. and Nilsson, N. J. (1971), STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, *Artificial Intelligence*, volume 2(2).
- Finney, S., Gardiol, N. H., Kaelbling, L. P. and Oates, T. (2002a), Learning with Deictic Representation, *Technical Report AI MEMO 2002-006*, MIT AI Lab, Cambridge, Massachusetts.
- (2002b), The Thing that We Tried Didn't Work very Well: Deictic Representations in Reinforcement Learning, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 154–161.
- Finton, D. J. (2002), *Cognitive Economy and the Role of Representation in On-line Learning*, Ph.D. thesis, University of Wisconsin, Madison.
- (2005), When do Differences matter? On-line Feature Extraction through Cognitive Economy, *Cognitive Systems Research*, volume 6, pp. 263–281.
- Finzi, A. and Lukasiewicz, T. (2004a), Game-Theoretic Agent Programming in Golog, in: *Proceedings of the European Conference on Artificial Intelligence (ECAI)*.
- (2004b), Game-Theoretic Agent Programming in Golog, *Technical Report INFSYS Research Report 1843-04-02*, Technical University of Vienna, Austria.
- (2004c), Relational Markov Games, in: *European Conference on Logics in Artificial Intelligence (JELIA)*, volume 3229 of *Lecture Notes in Computer Science*, pp. 320–333.
- Fischer, J. (2005), *Asynchrone Relationale Werte Iteration*, Master's thesis, Albert-Ludwigs-University, Freiburg, Germany, in German.
- Fitch, R., Hengst, B., Suc, D., Calbert, G. and Scholz, J. (2005), Structural Abstraction Experiments in Reinforcement Learning, in: *Australian Conference on Artificial Intelligence*, volume 3809 of *Lecture Notes in Computer Science*, pp. 164–175.
- Flach, P. A. (1994), *Simply Logical: Intelligent Reasoning by Example*, John Wiley.
- Flach, P. A. and Lachiche, N. (1999), 1BC: A First-Order Bayesian Classifier, in: *Proceedings of the International Conference on Inductive Logic Programming (ILP)*, volume 1634 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Berlin, pp. 14–27.
- Fodor, J. A. and Pylyshyn, Z. W. (1988), Connectionism and Cognitive Architecture: A Critical Analysis, *Cognition*, volume 28, pp. 3–71, reprinted in *Mind Design II: Philosophy, Psychology, Artificial Intelligence*

BIBLIOGRAPHY

- (1997) by J. Haugeland (ed.), The MIT Press, Cambridge, Massachusetts.
- Främling, K. (2005), Bi-Memory Model for Guiding Exploration by Pre-Existing Knowledge, in: Driessens, K., Fern, A. and van Otterlo, M. (eds.), *Proceedings of the ICML-2005 Workshop on Rich Representations for Reinforcement Learning*, pp. 21–26.
- Frege, G. (1879), *Begriffsschrift, Eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens*, Halle.
- French, R. M. (1999), Catastrophic Forgetting in Connectionist Networks, *Trends in Cognitive Science*, volume 3(4).
- Friedman, N. (1998), The Bayesian Structural EM algorithm, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 129–138.
- Fritzke, B. (1994), Growing Cell Structures – A Self-Organizing Network for Unsupervised and Supervised Learning, *Neural Networks*, volume 7, pp. 1441–1460.
- (1995), A Growing Neural Gas Network learns Topologies, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 625–632.
- (1997), Unsupervised Ontogenic Networks, in: Fiesler, E. and Beale, R. (eds.), *Handbook of Neural Computation*, chapter C2.4, Institute of Physics and Oxford University Press, New York, New York.
- Frühwirth, T. W. (1998), Theory and Practice of Constraint Handling Rules, *Journal of Logic Programming*, volume 37(1–3), pp. 95–138.
- Gabaldon, A. (2000), Non-Markovian Control in the Situation Calculus, in: *Proceedings of the Second International Cognitive Robotics Workshop*.
- Galavotti, M. C. (2005), *Philosophical Introduction to Probability*, CSLI Publications, United States.
- Gamut, L. T. F. (1991), *Logic, Language and Meaning, Volume 2: Intensional Logic and Logical Grammar*, The University of Chicago Press, Chicago.
- García-Durán, R., Fernández, F. and Borrajo, D. (2008), Learning and Transferring Relational Instance-Based Policies, in: *Proceedings of the AAIL-2008 Workshop on Transfer Learning for Complex Tasks*.
- Gärdenfors, P. (1988), *Knowledge in Flux: Modeling the Dynamics of Epistemic States*, The MIT Press, Bradford Books, Cambridge, Massachusetts.
- (2000), *Conceptual Spaces: The Geometry of Thought*, The MIT Press, Cambridge, Massachusetts.
- Gardiol, N. H. (2003), *Applying Probabilistic Rules to Relational Worlds*, Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.
- (2007), *Relational Envelope-based Planning*, Ph.D. thesis, MIT, Cambridge, MA.
- Gardiol, N. H. and Kaelbling, L. P. (2003), Envelope-based Planning in Relational MDPs, in: *Proceedings of the Neural Information Processing Conference (NIPS)*.
- (2006a), Computing Action Equivalences for Planning, in: *International Conference on Automated Planning and Scheduling, Doctoral Consortium*, Cumbria, UK.
- (2006b), Computing Action Equivalences under Time-Constraints, *Technical Report AIM-2006-022*, MIT Computer Science and Artificial Intelligence Laboratory (CSAIL).
- (2007), Action-Space Partitioning for Planning, in: *National Conference on Artificial Intelligence (AAAI)*, Vancouver, Canada.
- (2008), Adaptive Envelope MDPs for Relational Equivalence-based Planning, *Technical Report MIT-CSAIL-TR-2008-050*, MIT CS & AI Lab, Cambridge, MA.
- Gärtner, T. (2003), A Survey of Kernels for Structured Data, *SIGKDD Explorations*, volume 5, pp. 49–58.
- Gärtner, T., Driessens, K. and Ramon, J. (2003), Graph Kernels and Gaussian Processes for Relational Reinforcement Learning, in: *Proceedings of the International Conference on Inductive Logic Programming (ILP)*.
- Gearhart, C. (2003), Genetic Programming as Policy Search in Markov Decision Processes, in: *Genetic Algorithms and Genetic Programming at Stanford*, pp. 61–67.
- Geffner, H. (1998), Modeling Intelligent Behavior: The Markov Decision Process Approach, in: *Progress in Artificial Intelligence*, volume 1484 of *Lecture Notes in Artificial Intelligence*, pp. 1–12.
- (2000), Functional STRIPS, in: Minker (2000b), chapter 9, pp. 187–209.
- Geffner, H. and Bonet, B. (1998), High-Level Planning and Control with Incomplete Information using POMDPs, in: *Proceedings Fall AAIL Symposium on Cognitive Robotics*.
- Gelfond, M. and Lifschitz, V. (1998), Action Languages, *Electronic Transactions on Artificial Intelligence*, vol-

BIBLIOGRAPHY

- ume 2, pp. 193–210.
- Gérard, P., Meyer, J. A. and Sigaud, O. (2005), Combining Latent Learning with Dynamic Programming in the Modular Anticipatory Classifier System, *European Journal of Operational Research*, volume 160(3), pp. 614–637.
- Gérard, P., Stolzmann, W. and Sigaud, O. (2002), YACS: A New Learning Classifier System with Anticipation, *Journal of Soft Computing: Special Issue on Learning Classifier Systems*, volume 6, pp. 216–228.
- Getoor, L., Friedman, N., Koller, D. and Taskar, B. (2001), Probabilistic Models of Relational Structure, in: *Proceedings of the International Conference on Machine Learning (ICML)*.
- Getoor, L. and Jensen, D. (eds.) (2004), *Proceedings of the ICML-2004 Workshop on Learning Statistical from Relational Data (SRL'03)*.
- Getoor, L. and Taskar, B. (eds.) (2007), *An Introduction to Statistical Relational Learning*, The MIT Press, Cambridge, Massachusetts.
- Ghahramani, Z. (1998), Learning Dynamic Bayesian Networks, in: Giles, C. L. and Gori, M. (eds.), *Adaptive Processing of Sequence and Data Structures*, pp. 168–197.
- Ghallab, M., Howe, A. E., Knoblock, C. A., McDermott, D. V., Ram, A., Veloso, M., Weld, D. S. and Wilkins, D. (1998), PDDL - The Planning Domain Definition Language, *Technical Report CVC TR-98-003/DCS TR-1165*, Yale Center for Computational Vision and Control.
- Ghavamzadeh, M. and Mahadevan, S. (2003), Hierarchical Policy Gradient Algorithms, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 226–233.
- Gil, Y. (1994), Learning by Experimentation: Incremental Refinement of Incomplete Planning Domains, in: *Proceedings of the International Conference on Machine Learning (ICML)*.
- Giunchiglia, F. and Walsh, T. (1992), A Theory of Abstraction, *Artificial Intelligence*, volume 57, pp. 323–389.
- Givan, R. and Dean, T. (1997), Model Minimization, Regression, and Propositional STRIPS planning, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.
- Givan, R., Dean, T. and Greig, M. (2003), Equivalence Notions and Model Minimization in Markov Decision Processes, *Artificial Intelligence*, volume 147, pp. 163–223.
- Glaubius, R., Namihira, M. and Smart, W. D. (2005), Speeding up Reinforcement Learning using Manifold Representations: Preliminary Results, in: *Proceedings of the Workshop Reasoning with Uncertainty in Robotics at IJCAI'05*.
- Goetschalckx, R. and Driessens, K. (2007), Cost-Sensitive Reinforcement Learning, in: *Workshop on Artificial Intelligence Planning and Learning at the International Conference on Automated Planning Systems*.
- Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley, Reading, MA.
- Goldbloom Bloch, W. (2008), *The Unimaginable Mathematics of Borges' Library of Babel*, Oxford University Press.
- Goldsmith, J. and Sloan, R. H. (2000), The Complexity of Model Aggregation, in: *Proceedings of the Conference on AI and Planning Systems (AIPS)*.
- Gordon, G. J. (1995a), Online Fitted Reinforcement Learning, in: *The ICML Workshop on Value Function Approximation*.
- (1995b), Stable Function Approximation in Dynamic Programming, *Technical Report CMU-CS-95-103*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.
- (1995c), Stable Function Approximation in Dynamic Programming, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 261–268.
- (2000), Reinforcement Learning with Function Approximation Converges to a Region, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 1040–1046.
- Görtz, G., Rollinger, C. R. and Schneeberger, J. (eds.) (2003), *Handbuch der Künstlichen Intelligenz*, Oldenbourg Verlag, München, Wien, 4th edition, (in German).
- Gottlob, G. and Fermüller, C. G. (1993), Removing Redundancy from a Clause, *Artificial Intelligence*, volume 61, pp. 263–289.
- Grant, T. J. (1996), *Inductive Learning of Knowledge-Based Planning Operators*, Ph.D. thesis, Universiteit Maastricht, The Netherlands.
- Greiner, R. (1999), The Complexity of Theory Revision, *Artificial Intelligence*, volume 107, pp. 175–217.
- Gretton, C. (2007a), Gradient-Based Relational Reinforcement-Learning of Temporally Extended Policies, in:

- Proceedings of the International Conference on Artificial Intelligence Planning Systems (ICAPS)*.
- (2007b), Gradient-Based Relational Reinforcement Learning of Temporally Extended Policies, in: *Workshop on Artificial Intelligence Planning and Learning at the International Conference on Automated Planning Systems*.
- Gretton, C. and Thiébaux, S. (2004a), Exploiting First-Order Regression in Inductive Policy Selection, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 217–225.
- (2004b), Exploiting First-Order Regression in Inductive Policy Selection (Extended Abstract), in: *Proceedings of the Workshop on Relational Reinforcement Learning at ICML'04*.
- Groote, J. F. and Tveretina, O. (2003), Binary Decision Diagrams for First-Order Predicate Logic, *The Journal of Logic and Algebraic Programming*, volume 57, pp. 1–22.
- Grosskreutz, H. and Lakemeyer, G. (2000), Turning High-Level Plans into Robot Programs in Uncertain Domains, in: *Proceedings of the European Conference on Artificial Intelligence (ECAI)*.
- Großmann, A. (2000), Adaptive State-Space Quantisation and Multi-Task Reinforcement Learning Using Constructive Neural Networks, in: *From Animals to Animats: Proceedings of The International Conference on Simulation of Adaptive Behavior (SAB)*, pp. 160–169.
- (2001), *Continual Learning for Mobile Robots*, Ph.D. thesis, School of Computer Science, University of Birmingham, UK.
- Großmann, A., Hölldobler, S. and Skvortsova, O. (2002), Symbolic Dynamic Programming within the Fluent Calculus, in: *Proceedings of the IASTED International Conference on Artificial and Computational Intelligence*, pp. 378–383.
- Großmann, A. and Poli, R. (1997), Continual Robot Learning with Constructive Neural Networks, *Technical Report CSRP-97-11*, University of Birmingham, School of Computer Science.
- Grounds, M. and Kudenko, D. (2005), Combining Reinforcement Learning with Symbolic Planning, in: *Proceedings of the Fifth European Workshop on Adaptive Agents and Multi-Agent Systems*.
- Gu, D. and Yang, E. (2004), Multiagent Reinforcement Learning for Multi-Robot Systems: A Survey, *Technical Report CSM-404*, Department of Computer Science, University of Essex.
- Gu, Y. (2003), Macro-Actions in the Stochastic Situation Calculus, in: *Proceedings of IJCAI Workshop on Nonmonotonic Reasoning, Action, and Change*.
- Guestrin, C. (2003), *Planning Under Uncertainty in Complex Structured Environments*, Ph.D. thesis, Computer Science Department, Stanford University.
- Guestrin, C., Koller, D., Gearhart, C. and Kanodia, N. (2003a), Generalizing Plans to New Environments in Relational MDPs, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1003–1010.
- Guestrin, C., Koller, D., Parr, R. and Venkataraman, S. (2003b), Efficient Solution Algorithms for Factored MDPs, *Journal of Artificial Intelligence Research (JAIR)*, volume 19, pp. 399–468.
- Guo, H. F. and Gupta, G. (2004), Simplifying Dynamic Programming via Tabling, in: *Conference on Practical Aspects of Declarative Languages (PADL)*, volume 3057 of *Lecture Notes in Computer Science*, pp. 163–177.
- Gupta, N. and Nau, D. S. (1992), On the Complexity of Blocks-World Planning, *Artificial Intelligence*, volume 56, pp. 223–254.
- Haddon, M. (2003), *The Curious Incident of the Dog in the Night-Time*, Vintage, London.
- Halbritter, F. and Geibel, P. (2007), Learning Models of Relational MDPs Using Graph Kernels, in: *Proceedings of the Mexican Conference on Artificial Intelligence (MICAI)*, pp. 409–419.
- Halpern, J. Y. (1990), An Analysis of First-Order Logics of Probability, *Artificial Intelligence*, volume 46, pp. 311–350.
- (2003), *Reasoning about Uncertainty*, The MIT Press, Cambridge, Massachusetts.
- Hamker, F. H. (2001), Life-Long Learning Cell Structures - Continuously Learning without Catastrophic Interference, *Neural Networks*, volume 14, pp. 551–573.
- Hanks, S. and McDermott, D. V. (1994), Modeling a Dynamic and Uncertain World I: Symbolic and Probabilistic reasoning about change, *Artificial Intelligence*, volume 66(1), pp. 1–55.
- Hansen, E. A. and Zilberstein, S. (2001), LAO*: A Heuristic Search Algorithm that Finds Solutions with Loops, *Artificial Intelligence*, volume 129, pp. 35–62.
- Hanson, S. J. (1990), Meiosis networks, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 533–541.

BIBLIOGRAPHY

- Harnad, S. (1990), The Symbol Grounding Problem, *Physica D*, volume 42, pp. 335–346.
- Hastie, T., Tibshirani, R. and Friedman, J. H. (2001), *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer Verlag, New York.
- Haugeland, J. (ed.) (1997), *Mind Design II: Philosophy, Psychology and Artificial Intelligence*, A Bradford Book, The MIT Press, Cambridge, Massachusetts, revised and enlarged edition.
- Haykin, S. (1999), *Neural networks: a comprehensive foundation*, Prentice Hall, New Jersey, USA.
- Heinke, D. and Hamker, F. H. (1998), Comparing Neural Networks: A Benchmark on Growing Neural Gas, Growing Cell Structures and Fuzzy ARTMAP, *IEEE Transactions on Neural Networks*, volume 9(6), pp. 1279–1291.
- Helmert, M. (2003), Complexity for Standard Benchmark Domains in Planning, *Artificial Intelligence*, volume 143, pp. 219–262.
- Hengst, B. (2002), Discovering Hierarchy in Reinforcement Learning with HEXQ, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 243–250.
- (2003), Safe State Abstraction and Discounting in Hierarchical Reinforcement Learning, *Technical Report UNSW CSE TR 0308*, Computer Science and Engineering, University of New South Wales, Sidney, Australia.
- (2004), Model approximation for HEXQ hierarchical reinforcement learning, in: *Proceedings of the European Conference on Machine Learning (ECML)*, pp. 144–155, LNAI-3201.
- Hernandez, A. G., El Fallah-Seghrouchni, A. and Soldano, H. (2004), Learning in BDI Multi-agent Systems, in: *Proceedings of CLIMA IV - Computational Logic in Multi-Agent Systems*.
- Hertzberg, J. and Saffiotti, A. (2008), Editorial: Using Semantic Knowledge in Robotics, *Robotics and Autonomous Systems*, volume 56, pp. 875–877.
- Hindriks, K. V., de Boer, F. S., van der Hoek, W. and Meyer, J. J. (1999), Agent Programming in 3APL, *Autonomous Agents and Multi-Agent Systems*, volume 2(4), pp. 357–401.
- Hitzler, P., Hölldobler, S. and Seda, A. K. (2004), Logic Programs and Connectionist Networks, *Journal of Applied Logic*, volume 2, pp. 245–272.
- Hoey, J., St-Aubin, R., Hu, A. and Boutilier, C. (1999), SPUDD: Stochastic Planning using Decision Diagrams, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 279–288.
- (2000), Optimal and Approximate Stochastic Planning using Decision Diagrams, *Technical Report TR-00-05*, Department of Computer Science, University of British Columbia, Vancouver, BC, Canada.
- Hofstadter, D. R. (1979), *Gödel, Escher, Bach: an Eternal golden Braid*, Harvester Press Ltd., Great Britain.
- Holland, J. H. (1975), *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*, University of Michigan Press, Ann Arbor, MI.
- (1986), Escaping Brittleness: The Possibilities of General Purpose Algorithms applied to Parallel Rule-Based Systems, in: Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (eds.), *Machine Learning, an Artificial Intelligence approach*, volume 2, Morgan Kaufmann, San Mateo, California, pp. 593–623.
- Hölldobler, S., Karabaev, E. and Skvortsova, O. (2006), FLUCAP: A Heuristic Search Planner for First-Order MDPs, *Journal of Artificial Intelligence Research (JAIR)*, volume 27, Engineering Note.
- Hölldobler, S. and Skvortsova, O. (2004a), A Logic-based Approach to Dynamic Programming, in: *Proceedings of the AAI Workshop on Learning and Planning in Markov Processes - Advances and Challenges*.
- (2004b), A Normalization Algorithm for Automated First-Order Value Iteration, Technical Report.
- Holmes, M. P. and Isbell jr., C. L. (2004), Schema Learning: Experience-Based Construction of Predictive Action Models, in: *Proceedings of the Neural Information Processing Conference (NIPS)*.
- Holte, R. C. and Choueiry, B. Y. (2003), Abstraction and Reformulation in Artificial Intelligence, *Phil. Trans. R. Soc. Lond. B.*, volume 358, pp. 1197–1204.
- Howard, R. A. (1960), *Dynamic Programming and Markov Processes*, The MIT Press, Cambridge, Massachusetts.
- Irodova, M. and Sloan, R. H. (2005), Reinforcement Learning and Function Approximation, in: *Proceedings of the International Florida Artificial Intelligence Research Society Conference (FLAIRS)*.
- Itoh, H. and Nakamura, K. (2004), Towards Learning to Learn and Plan by Relational Reinforcement Learning, in: *Proceedings of the ICML Workshop on Relational Reinforcement Learning*.
- Jacobs, S., Ferrein, A. and Lakemeyer, G. (2005), Unreal Golog Bots, in: *Proceedings of the IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*.

- Jain, A. K. (1989), *Fundamentals of Digital Image Processing*, Prentice-Hall, Englewood Cliffs, NJ.
- Jain, A. K., Hong, L., Pankanti, S. and Bolle, R. (1997), An Identity-Authentication System Using Fingerprints, *Proceedings of the IEEE*, volume 85(9), pp. 1365–1388.
- Jang, J. S. R. (1993), ANFIS: Adaptive-Network-Based Fuzzy Inference System, *IEEE Transactions on Systems, Man, and Cybernetics*, volume 23(3), pp. 665–685.
- Jang, J. S. R., Sun, C. T. and Mizutani, E. (1997), *Neuro-Fuzzy and Soft Computing*, Prentice Hall.
- Jiang, X., Yau, W. Y. and Ser, W. (2001), Detecting the Fingerprint Minutiae by Adaptive Tracing the Gray-Level Ridge, *Pattern Recognition*, volume 34(5), pp. 999–1013.
- Jiménez, S., Fernández, F. and Borrajo, D. (2006), Inducing Non-Deterministic Actions Behaviour to Plan Robustly in Probabilistic Domains, in: *Workshop on Planning under Uncertainty and Execution Control for Autonomus Systems at ICAPS-2006*, pp. 67–74.
- Jong, N. K. and Stone, P. (2004), Towards Learning to Ignore Irrelevant State Variables, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.
- Jonsson, A. (2006), *A Causal Approach to Hierarchical Decomposition in Reinforcement Learning*, Ph.D. thesis, Department of Computer Science, University of Massachusetts, Amherst.
- Jonsson, A. and Barto, A. G. (2001), Automated State Abstraction for Options using the U-Tree Algorithm, in: *Proceedings of the Neural Information Processing Conference (NIPS)*.
- (2005), A Causal Approach to Hierarchical Decomposition of Factored MDPs, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 401–408.
- Jordan, M. I. (ed.) (1999), *Learning in Graphical Models*, The MIT Press, Cambridge, Massachusetts.
- Joshi, S. and Kharden, R. (2008), Stochastic Planning with First Order Decision Diagrams, in: *Proceedings of the International Conference on Artificial Intelligence Planning Systems (ICAPS)*.
- Joshi, S., Kharden, R. and Wang, C. (2006), First Order Decision Diagrams for Relational MDPs, in: *Proceedings of the ICML-06 Workshop on Open Problems in Statistical Relational Learning*.
- Jozefowicz, J. (2002), Reinforcement Learning and Conditioning: An Overview, unpublished.
- Kadie, C. M. (1988), Diffy-S: Learning Robot Operator Schemata from Examples, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 430–436.
- Kaelbling, L. P. (1993a), Hierarchical Learning in Stochastic Domains: Preliminary Results, in: *Proceedings of the International Conference on Machine Learning (ICML)*.
- (1993b), *Learning in Embedded Systems*, The MIT Press, Cambridge, Massachusetts.
- Kaelbling, L. P., Littman, M. L. and Cassandra, A. R. (1998), Planning and Acting in Partially Observable Stochastic Domains, *Artificial Intelligence*, volume 101, pp. 99–134.
- Kaelbling, L. P., Littman, M. L. and Moore, A. W. (1996), Reinforcement Learning: A Survey, *Journal of Artificial Intelligence Research*, volume 4, pp. 237–285.
- Kaelbling, L. P., Oates, T., Gardiol, N. H. and Finney, S. (2001), Learning in Worlds with Objects, in: *The AAAI Spring Symposium*.
- Kaikhah, K. and Garlick, R. (2000), Variable Hidden Layer Sizing in Elman Recurrent Neuro-Evolution, *Applied Intelligence*, volume 12, pp. 193–205.
- Kakade, S. M. (2003), *On the Sample Complexity of Reinforcement Learning*, Ph.D. thesis, Gatsby Computational Neuroscience Unit, University College London, Great Britain.
- Karabaev, E., Rammé, G. and Skvortsova, O. (2006), Efficient Symbolic Reasoning for First-Order MDPs, in: *ECAI Workshop on Planning, Learning and Monitoring with Uncertainty and Dynamic Worlds*.
- Karabaev, E. and Skvortsova, O. (2004), FCPlanner: A Planning Strategy for First-Order MDPs, ICAPS'04 Planning Competition, Probabilistic Track.
- (2005), A Heuristic Search Algorithm for Solving First-Order MDPs, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*.
- Karalic, A. and Bratko, I. (1997), First-Order Regression, *Machine Learning*, volume 26, pp. 147–176.
- Kasabov, N. K. (1998), The ECOS Framework and the ECO Learning Method for Evolving Connectionist Systems, *Journal of Advanced Computational Intelligence*, volume 2(6).
- (2001), On-Line Learning, Reasoning, Rule Extraction and Aggregation in Locally Optimized Evolving Fuzzy Neural Network, *Neurocomputing*, volume 41, pp. 25–45.
- Katz, D. and Brock, O. (2008), Manipulating Articulated Objects With Interactive Perception, in: *Proceedings of the IEEE International Conference on Robotics and Automation*.

BIBLIOGRAPHY

- Katz, D., Pyuro, Y. and Brock, O. (2008), Learning to Manipulate Articulated Objects in Unstructured Environments Using a Grounded Relational Representation, in: *Proceedings of Robotics: Science and Systems IV*.
- Kearns, M. and Koller, D. (1999), Efficient Reinforcement Learning for Factored MDPs, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 740–747.
- Kearns, M. and Singh, S. (1998), Near-Optimal Reinforcement Learning in Polynomial Time, in: *Proceedings of the International Conference on Machine Learning (ICML)*.
- Keerthi, S. S. and Ravindran, B. (1997), Reinforcement Learning, in: Fiesler, E. and Beale, R. (eds.), *Handbook of Neural Computation*, chapter C3, Institute of Physics and Oxford University Press, New York, New York.
- Keller, P. W., Mannor, S. and Precup, D. (2006), Automatic Basis Function Construction for Approximate Dynamic Programming and Reinforcement Learning, in: *Proceedings of the International Conference on Machine Learning (ICML)*.
- Kersting, K. and De Raedt, L. (2001), Towards Combining Inductive Logic Programming and Bayesian Networks, in: *Proceedings of the International Conference on Inductive Logic Programming (ILP)*.
- (2003), Logical Markov Decision Programs, in: *Proceedings of the IJCAI'03 Workshop on Learning Statistical Models of Relational Data*.
- (2004), Logical Markov Decision Programs and the Convergence of TD(λ), in: *Proceedings of the International Conference on Inductive Logic Programming (ILP)*.
- Kersting, K. and Driessens, K. (2008), Non-Parametric Gradients: A Unified Treatment of Propositional and Relational Domains, in: *Proceedings of the International Conference on Machine Learning (ICML)*.
- Kersting, K., Plagemann, C., Cocora, A., Burgard, W. and De Raedt, L. (2007), Learning to Transfer Optimal Navigation Policies, *Advanced Robotics. Special Issue on Imitative Robots*, volume 21(9).
- Kersting, K., Raiko, T., Kramer, S. and De Raedt, L. (2003), Towards Discovering Structural Signatures of Protein Folds based on Logical Hidden Markov Models, in: *Proceedings of the Pacific Symposium on Biocomputing*, pp. 192–203.
- Kersting, K., van Otterlo, M. and De Raedt, L. (2004), Bellman goes Relational, in: *Proceedings of the International Conference on Machine Learning (ICML)*.
- Khardon, R. (1999a), Learning Action Strategies for Planning Domains, *Artificial Intelligence*, volume 113, pp. 125–148.
- (1999b), Learning to Take Actions, *Machine Learning*, volume 35(1), pp. 57–90.
- Khoshafian, S. and Copeland, G. (1986), Object Identity, in: *Proceedings of the 1st ACM OOPSLA conference*, pp. 406–416.
- Kietz, J. U. and Lübke, M. (1994), An Efficient Subsumption Algorithm for Inductive Logic Programming, in: Cohen, W. and Hirsch, H. (eds.), *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 130–138.
- Kijsirikul, N., Sinthupinyo, S. and Chongkasemwongse, K. (2001), Approximate Match of Rules Using Back-propagation Neural Networks, *Machine Learning*, volume 44, pp. 273–299.
- Kim, K. E. and Dean, T. (2003), Solving Factored MDPs using Non-Homogeneous Partitions, *Artificial Intelligence*, volume 147, pp. 225–251.
- Kirsten, M., Wrobel, S. and Horváth, T. (2001), Distance Based Approaches to Relational Learning and Clustering, in: Džeroski and Lavrac (2001b), chapter 9, pp. 213–232.
- Kochenderfer, M. J. (2003), Evolving Hierarchical and Recursive Teleo-Reactive Programs through Genetic Programming, in: *EuroGP 2003*, volume 2610 of *Lecture Notes in Computer Science*, pp. 83–92.
- Koenig, S. and Liu, Y. (2002), The Interaction of Representations and Planning Objectives for Decision-Theoretic Planning, *Journal of Experimental and Theoretical Artificial Intelligence*, volume 14(4), pp. 303–326.
- Kok, J. R. (2006), *Coordination and Learning in Cooperative Multiagent Systems*, Ph.D. thesis, Universiteit van Amsterdam.
- Kok, S. and Domingos, P. (2005), Learning the Structure of Markov Logic Networks, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 441–448.
- Kokar, M. M. and Reveliotis, S. A. (1993), Reinforcement Learning: Architectures and Algorithms, *International Journal of Intelligent Systems*, volume 8, pp. 875–894.

- Koller, D. and Parr, R. (1999), Computing Factored Value Functions for Policies in Structured Domains, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1332–1339.
- (2000), Policy Iteration for Factored MDPs, in: *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI'00)*, pp. 326–334.
- Konda, V. and Tsitsiklis, J. (2003), Actor-Critic Algorithms, *SIAM Journal on Control and Optimization*, volume 42(4), pp. 1143–1166.
- Konidaris, G. (2006), A Framework for Transfer in Reinforcement Learning, in: *ICML-06 Workshop on Structural Knowledge Transfer for Machine Learning*.
- Korf, R. E. (1980), Toward a Model of Representation Changes, *Artificial Intelligence*, volume 14, pp. 41–78.
- Kraft, D., Bašeski, E., Popović, M., Batog, A. M., Kjær-Nielsen, A., Krüger, N., Petrick, R. P. A., Geib, C., Pugeault, N., Steedman, M., Asfour, T., Dillmann, R., Kalkan, S., Wörgötter, F., Hommel, B., Detry, R. and Piater, J. (2008), Exploration and Planning in a Three-Level Cognitive Architecture, in: *Proceedings of the International Conference on Cognitive Systems (CogSys)*, pp. 71–78.
- Kramer, S., Lavrac, N. and Flach, P. A. (2001), Propositionalization Approaches to Relational Data Mining, in: Džeroski and Lavrac (2001b), chapter 11, pp. 262–291.
- Kramer, S. and Widmer, G. (2001), Inducing Classification and Regression Trees in First Order Logic, in: Džeroski and Lavrac (2001b), chapter 6, pp. 140–159.
- Kress, M. and Seese, D. (2007), Executable Product Models – The Intelligent Way, in: *IEEE International Conference on Systems, Man and Cybernetics*, pp. 1987–1992.
- Kretschmar, R. and Anderson, C. W. (1997), Comparison of CMACs and Radial Basis Functions for Local Function Approximators in Reinforcement Learning, in: *Proceedings of the IEEE International Conference on Neural Networks*, pp. 834–837.
- Kripke, S. A. (1963), Semantical Considerations on Modal Logic, *Acta Philosophica Fennica, Modal and Many-Valued Logics*, pp. 83–94.
- Krueger, V., Kragic, D., Ude, A. and Geib, C. (2007), The Meaning of Action: A Review on action recognition and mapping, *Advanced Robotics*, volume 21(13), pp. 1473–1501.
- Kushmerick, N., Hanks, S. and Weld, D. S. (1995), An Algorithm for Probabilistic Planning, *Artificial Intelligence*, volume 76(1–2), pp. 239–286.
- Kwok, T. Y. and Yeung, D. Y. (1997), Constructive Algorithms for Structure Learning in Feedforward Neural Networks for Regression Problems, *IEEE Transactions on Neural Networks*, volume 8(3), pp. 630–645.
- Lagoudakis, M. G. and Parr, R. (2003), Reinforcement Learning as Classification: Leveraging Modern Classifiers, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 424–431.
- Lambert III, T. J., Epelman, M. A. and Smith, R. L. (2004), Aggregation in Stochastic Dynamic Programming, *Technical report*, University of Michigan.
- Lane, T., Ridens, M. and Stevens, S. (2007), Reinforcement Learning in Nonstationary Environment Navigation Tasks, in: *Proceedings on the Canadian Society for Computational Studies of Intelligence*, volume 4509 of *Lecture Notes in Computer Science*, Springer, pp. 429–440.
- Lane, T. and Wilson, A. (2005), Toward a Topological Theory of Relational Reinforcement Learning for Navigation Tasks, in: *Proceedings of the International Florida Artificial Intelligence Research Society Conference (FLAIRS)*.
- Lang, J., Lin, F. and Marquis, P. (2003), Causal Theories of Action: A Computational Core, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.
- Langford, J. and Zadrozny, B. (2005), Relating Reinforcement Learning to Classification Performance, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 473–480.
- Langley, P. (1996), *Elements of Machine Learning*, Morgan Kaufmann, San Francisco.
- (2006), Cognitive Architectures and General Intelligent Systems, *AI Magazine*, volume 27, pp. 33–44.
- Langley, P., Arai, S. and Shapiro, D. (2004), Model-Based Learning with Hierarchical Relational Skills, in: *Proceedings of the Workshop on Relational Reinforcement Learning at ICML'04*.
- Langley, P., Laird, J. E. and Rogers, S. (2006), Cognitive Architectures: Research Issues and Challenges, unpublished Draft.
- Lanzi, P. L. (2000), Adaptive Agents with Reinforcement Learning and Internal Memory, in: *From Animals to Animats: Proceedings of The International Conference on Simulation of Adaptive Behavior (SAB)*, pp. 333–342.

BIBLIOGRAPHY

- (2002), Learning Classifier Systems from a Reinforcement Learning Perspective, *Soft Computing*, volume 6, pp. 162–170.
- Lanzi, P. L., Stolzmann, W. and Wilson, S. W. (eds.) (2000), volume 1813 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Berlin.
- Lavrac, N. and Džeroski, S. (1994), *Inductive Logic Programming*, Ellis Harwood, New York.
- Lawrence, S., Tsoi, A. C. and Back, A. D. (1996), Function Approximation with Neural Networks and Local Methods: Bias, Variance and Smoothness, in: Bartlett, P., Burkitt, A. and Williamson, R. (eds.), *Australian Conference on Neural Networks*, Australian National University, pp. 16–21.
- Lecoeuche, R. (2001), Learning Optimal Dialogue Management Rules by Using Reinforcement Learning and Inductive Logic Programming, in: *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*.
- Lehtokangas, M. (1999), Modeling with Constructive Backpropagation, *Neural Networks*, volume 12(4–5), pp. 707–716.
- Letia, I. and Precup, D. (2001), Developing Collaborative Golog Agents by Reinforcement Learning, in: *Proceedings of the 13th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'01)*, IEEE Computer Society.
- Levesque, H. J., Reiter, R. and Lesperance, Y. (1997), Golog: A Logic Programming Language for Dynamic Domains, *Journal of Logic Programming*, volume 31(1–3), pp. 59–83.
- Levine, J. and Humphreys, D. (2003), Learning Action Strategies for Planning Domains using Genetic Programming, in: *EvoWorkshops 2003*, volume 2611 of *Lecture Notes in Computer Science*, pp. 684–695.
- Levner, I., Bulitko, V., Madani, O. and Greiner, R. (2002), Performance of Lookahead Control Policies in the Face of Abstractions and Approximations, *Lecture Notes in Computer Science*, volume 2371, pp. 299–307.
- Li, L. and Littman, M. L. (2005), Lazy Approximation for Solving Continuous Finite-Horizon MDPs, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.
- Li, L., Walsh, T. J. and Littman, M. L. (2006), Towards a Unified Theory of State Abstraction for MDPs, in: *Proceedings of AI-MATH*.
- Liao, L., Fox, D. and Kautz, H. (2005), Location-based Activity Recognition, in: *Proceedings of the Neural Information Processing Conference (NIPS)*.
- Lifschitz, V. (1986), On the Semantics of STRIPS, in: Georgeff, M. and Lanskey, A. (eds.), *Reasoning about Actions and Plans*, pp. 1–9.
- Lifschitz, V. (ed.) (1990), *Formalizing Common Sense: Papers by John McCarthy*, Ablex Publishing Company, Norwood.
- Lin, F. (2003), Compiling Causal Theories to Successor State Axioms and STRIPS-Like Systems, *Journal of Artificial Intelligence Research (JAIR)*, volume 19, pp. 279–314.
- Lin, K. J. (1992), Self-Improving Reactive Agents based on Reinforcement Learning, *Machine Learning*, volume 8, pp. 293–321.
- Littman, M. L. (1997), Probabilistic Propositional Planning: Representations and Complexity, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 748–754.
- Littman, M. L. and Boyan, J. A. (1993), A Distributed Reinforcement Learning Scheme for Network Routing, in: *Proceedings of the First International Workshop on Applications of Neural Networks to Telecommunication*, pp. 45–51.
- Littman, M. L., Dean, T. and Kaelbling, L. P. (1995), On the Complexity of Solving Markov Decision Problems, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 394–402.
- Littman, M. L., Diuk, C. and Strehl, A. (2005), A Hierarchical Approach to Efficient Reinforcement Learning, in: Driessens, K., Fern, A. and van Otterlo, M. (eds.), *Proceedings of the ICML'05 Workshop on Rich Representations for Reinforcement Learning*, pp. 33–38.
- Littman, M. L. and Majercik, S. M. (1997), Large-Scale Planning under Uncertainty: A Survey, in: *Workshop on Planning and Scheduling for Space*, pp. 1–8.
- Littman, M. L., Sutton, R. S. and Singh, S. (2001), Predictive Representations of State, in: *Proceedings of the Neural Information Processing Conference (NIPS)*.
- Lloyd, J. W. (1991), *Foundations of Logic Programming*, Springer-Verlag, Berlin, second edition.
- (2003), *Logic for Learning: Learning Comprehensible Theories From Structured Data*, Springer-Verlag.
- Lorenzo, D. and Otero, R. P. (2000), Using an ILP Algorithm to Learn Logic Programs for Reasoning about

BIBLIOGRAPHY

- Actions, in: Cussens, J. and Frisch, A. (eds.), *Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming*, pp. 163–171.
- Lozano-Perez, T., Mason, M. M. and Taylor, R. H. (1984), Automatic Synthesis of Fine-Motion Strategies for Robots, *International Journal of Robotics Research*, volume 3.
- Luger, G. F. (2002), *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, Addison-Wesley, London.
- Mahadevan, S. (1996), Average Reward Reinforcement Learning: Foundations, Algorithms, and Empirical Results, *Machine Learning*, volume 22, pp. 159–195.
- (2005a), Proto-Value Functions: Developmental Reinforcement Learning, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 553–560.
- (2005b), Representation Policy Iteration, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*.
- Mahadevan, S. and Maggioni, M. (2007), Proto-Value Functions: A Laplacian Framework for Learning Representation and Control in Markov Decision Processes, *Journal of Machine Learning Research (JMLR)*, volume 8, pp. 2169–2231.
- Maio, D. and Maltoni, D. (1997), Direct Gray-Scale Minutiae Detection in Fingerprints, *IEEE Transactions PAMI*, volume 19(1), pp. 27–39.
- Makino, T. and Takagi, T. (2008), On-line Discovery of Temporal-Difference Networks, in: *Proceedings of the International Conference on Machine Learning (ICML)*.
- Maloberti, J. and Sebag, M. (2004), Fast Theta-Subsumption with Constraint Satisfaction Algorithms, *Machine Learning*, volume 55, pp. 137–174.
- Maloo, M. A. (2003), Incremental Rule Learning with Partial Instance Memory for Changing Concepts, in: *Proceedings of the International Joint Conference on Neural Networks*, pp. 2764–2769.
- Manfredi, V. and Mahadevan, S. (2005), Hierarchical Reinforcement Learning using Graphical Models, in: Driessens, K., Fern, A. and van Otterlo, M. (eds.), *Proceedings of the ICML'05 Workshop on Rich Representations for Reinforcement Learning*, pp. 39–44.
- Manna, Z. and Waldinger, R. (1978), The Synthesis of Structure-Changing Programs, in: *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 175–187.
- Mannor, S., Menache, I., Hoze, A. and Klein, U. (2004), Dynamic Abstraction in Reinforcement Learning via Clustering, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 560–567.
- Mansour, Y. and Singh, S. (1999), On the complexity of Policy Iteration, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 401–408.
- Marecki, J., Topol, Z. and Tambe, M. (2006), A Fast Analytical Algorithm for MDPs with Continuous State Spaces, in: *AAMAS-06 Proceedings of 8th Workshop on Game Theoretic and Decision Theoretic Agents*.
- Margolis (1999), *Concepts*, The MIT Press, Cambridge, Massachusetts.
- Markman, A. B. (1999), *Knowledge Representation*, Lawrence Erlbaum Associates, Publishers, Mahwah, New Jersey.
- Markman, A. B. and Dietrich, E. (2000a), Extending the Classical View of Representation, *Trends in Cognitive Science*, volume 4(12), pp. 470–475.
- (2000b), In Defense of Representation, *Cognitive Psychology*, volume 40, pp. 138–171.
- Martin, M. and Geffner, H. (2000), Learning Generalized Policies in Planning using Concept Languages, in: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*.
- (2004), Learning Generalized Policies from Planning Examples using Concept Languages, *Applied Intelligence*, volume 20, pp. 9–19.
- Mataric, M. J. (1994), Reward Functions for Accelerated Learning, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 181–189.
- Matthews, W. H. (1922), *Mazes and Labyrinths: A General Account of their History and Developments*, Longmans, Green and Co., London, reprinted in 1970 by Dover Publications, New York, under the title 'Mazes & Labyrinths: Their History & Development'.
- Mausam and Weld, D. S. (2003), Solving Relational MDPs with First-Order Machine Learning, in: *Workshop on Planning under Uncertainty and Incomplete Information at ICAPS'03*.
- McAllester, D. A. (1999), World-Modeling vs. World-Axiomatizing, in: *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pp. 375–388.

BIBLIOGRAPHY

- (2000), Bellman Equations for Stochastic Programs, revision of talk at LPNMR-99.
- McCallum, A. K. (1996), *Reinforcement Learning with Selective Perception and Hidden State*, Ph.D. thesis, Department of Computer Science, University of Rochester, Rochester, New York.
- McCallum, R. A. (1995), Instance-Based Utile Distinctions for Reinforcement Learning with Hidden State, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 387–395.
- McCarthy, J. (1959), Programs with Common Sense, in: *Proceedings of the Teddington Conference on the Mechanisms of Thought Processes*, London: Her Majesty's Stationary Office, pp. 75–91, reprinted in Lifschitz 1989.
- (1963), Situations, Actions and Causal Laws, *Technical report*, Stanford University.
- (2000), Concepts of Logical AI, in: Minker, J. (ed.), *Logic-Based Artificial Intelligence*, Kluwer Academic Publishers Group, Dordrecht, The Netherlands.
- McCarthy, J. and Hayes, P. J. (1969), Some Philosophical Problems from the Standpoint of Artificial Intelligence, in: Meltzer, B. and Michie, D. (eds.), *Machine Intelligence 4*, Edinburgh University Press, Edinburgh, Scotland, pp. 463–502.
- McGovern, A. and Barto, A. G. (2001), Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density, in: *Proceedings of the International Conference on Machine Learning (ICML)*.
- McMahan, H. B., Likhachev, M. and Gordon, G. J. (2005), Bounded Real-Time Dynamic Programming: RTDP with Monotone Upper Bounds and Performance Guarantees, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 569–576.
- Mehta, N., Ray, S., Tadepalli, P. and Dietterich, T. G. (2008), Automatic Discovery and Transfer of MAXQ Hierarchies, in: *Proceedings of the International Conference on Machine Learning (ICML)*.
- Mellor, D. (2005a), A First Order Logic Classifier System, in: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*.
- (2005b), Policy Transfer with a Relational Learning Classifier System, in: *Proceedings of the Learning Classifier Workshop at GECCO'05*.
- (2007), *A Learning Classifier System Approach to Relational Reinforcement Learning*, Ph.D. thesis, School of Electrical Engineering and Computer Science, University of Newcastle, Australia, submitted version.
- (2008), A Learning Classifier System Approach to Relational Reinforcement Learning, in: *10th and 11th International Workshops on Learning Classifier Systems (IWLCS 2006–2007)*, Revised Selected Papers, volume 4998 of *Lecture Notes in Computer Science*, Springer, pp. 169–188.
- Menache, I., Mannor, S. and Shimkin, N. (2002), Q-Cut: Dynamic Discovery of Sub-Goals in Reinforcement Learning, in: *Proceedings of the European Conference on Machine Learning (ECML)*, pp. 295–306.
- Meuleau, N. and Peshkin, L. (1999), Off-Policy Policy Search, *Technical report*, MIT AI Lab, Massachusetts.
- Miller, W. T., Glanz, F. H. and Kraft, L. G. (1990), CMAC: An Associative Neural Network Alternative to Backpropagation, in: *Proceedings of the IEEE*, volume 78, pp. 1561–1567.
- Minker, J. (2000a), Introduction to Logic-Based Artificial Intelligence, in: Minker, J. (ed.), *Logic-Based Artificial Intelligence*, Kluwer Academic Publishers Group, Dordrecht, The Netherlands.
- (2000b), *Logic-Based Artificial Intelligence*, Kluwer Academic Publishers Group, Dordrecht, The Netherlands.
- Minsky, M. (1985), *Society of Mind*, Simon & Schuster, New York.
- Minsky, M. and Papert, S. A. (1988), *Perceptrons*, The MIT Press, Cambridge, Massachusetts, expanded Edition of the 1969 version.
- Minton, S., Carbonell, J., Knoblock, C. A., Kuokka, D. R., Etzioni, O. and Gil, Y. (1989), Explanation-Based Learning: A Problem Solving Perspective, *Artificial Intelligence*, volume 40(1–3), pp. 63–118.
- Mitchell, M. (1996), *An Introduction to Genetic Algorithms*, The MIT Press, Cambridge, Massachusetts.
- Mitchell, T. M. (1997), *Machine Learning*, McGraw-Hill, New York.
- Miyamoto, Y. and Uehara, K. (1999), Discovering New Features to Improve Q-Learning more Efficiently, *Technical Report CS24*, Department of Computer and Systems Engineering, Kobe University.
- Mooney, R. J. and Califf, M. E. (1995), Induction of First-Order Decision Lists: Results on Learning the Past Tense of English Verbs, *Journal of Artificial Intelligence Research (JAIR)*, volume 3, pp. 1–24.
- Moore, A. W. (1991), Variable Resolution Dynamic Programming: Efficiently Learning Action Maps in Multivariate Real-Valued State-Spaces, in: *Proceedings of the 8th International Workshop on Machine Learning*.

- Moore, A. W. and Atkeson, C. G. (1993), Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time, *Machine Learning*, volume 13(1), pp. 103–130.
- (1995), The Parti-game Algorithm for Variable Resolution Reinforcement Learning in Multidimensional State-spaces, *Machine Learning*, volume 21, pp. 199–233.
- Moore, A. W., Atkeson, C. G. and Schaal, S. (1995), Memory-Based Learning for Control, *Technical Report CMU-RI-TR-95-18*, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, USA.
- Morales, E. F. (2003), Scaling Up Reinforcement Learning with a Relational Representation, in: *Proceedings of the Workshop on Adaptability in Multi-Agent Systems at AORC'03*, Sydney.
- (2004a), Learning to Fly by Combining Reinforcement Learning with Behavioral Cloning, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 598–605.
- (2004b), Relational State Abstractions for Reinforcement Learning, in: *Proceedings of the Workshop on Relational Reinforcement Learning at ICML'04*.
- Moriarty, D. E., Handley, S. and Langley, P. (1998), Learning Distributed Strategies for Traffic Control, in: *From Animals to Animats: Proceedings of The International Conference on Simulation of Adaptive Behavior (SAB)*, pp. 437–446.
- Moriarty, D. E. and Miikkulainen, R. (1996), Efficient Reinforcement Learning through Symbiotic Evolution, *Machine Learning*, volume 22, pp. 11–32.
- (1998), Forming Neural Networks through Efficient and Adaptive Coevolution, *Evolutionary Computation*, volume 5(4), pp. 373–399.
- Moriarty, D. E., Schultz, A. C. and Grefenstette, J. J. (1999), Evolutionary Algorithms for Reinforcement Learning, *Journal of Artificial Intelligence Research (JAIR)*, volume 11, pp. 241–276.
- Mourão, K., Petrick, R. P. A. and Steedman, M. (2008), Using Kernel Perceptrons to Learn Action Effects for Planning, in: *Proceedings of the International Conference on Cognitive Systems (CogSys)*, pp. 45–50.
- Mueller, E. T. (2006), *Commonsense Reasoning*, Morgan Kaufman Publishers, Amsterdam.
- Muggleton, S. H. (1995), Inverse Entailment and Progol, *New Generation Computing Journal*, volume 13, pp. 245–286.
- Muggleton, S. H. and De Raedt, L. (1994), Inductive Logic Programming: Theory and Methods, *The Journal of Logic Programming*, volume 19 & 20, pp. 629–680.
- Muggleton, S. H. and Feng, C. (1992), Efficient Induction of Logic Programs, in: Muggleton, S. H. (ed.), *Inductive Logic Programming*, Academic Press.
- Muller, T. J. (2005), *Unintelligent Design: Evolutionary Algorithms in Relational Reinforcement Learning*, Master's thesis, Department of Computer Science, University of Twente, Enschede, The Netherlands.
- Muller, T. J. and van Otterlo, M. (2005), Evolutionary Reinforcement Learning in Relational Domains, in: *Proceedings of the 7th European Workshop on Reinforcement Learning*.
- Munos, R. and Moore, A. W. (1999), Variable Resolution Discretization for High-Accuracy Solutions of Optimal Control Problems, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1348–1355.
- (2002), Variable Resolution Discretization in Optimal Control, *Machine Learning*, volume 49, pp. 291–323.
- Murphy, K. (2000), A Survey of POMDP Solution Techniques, *Technical report*, Berkeley and California, California, California.
- Murtagh, S. and Utgoff, P. E. (2006), Solving a Problem Domain with Brute Force Goal Regression, *Technical Report UM-CS-2006-008*, Department of Computer Science, University of Massachusetts at Amherst.
- Nason, S. and Laird, J. E. (2004a), Soar-RL: Integrating Reinforcement Learning with Soar, in: *Proceedings of the Workshop on Relational Reinforcement Learning at ICML'04*.
- (2004b), Soar-RL: Integrating Reinforcement Learning with Soar, in: *Proceedings of the Sixth International Conference on Cognitive Modeling*, pp. 208–213.
- (2005), Soar-RL: Integrating Reinforcement Learning with Soar, *Cognitive Systems Research*, volume 6(1), pp. 51–59.
- Needham, C. J., Santos, P. E., Magee, D. R., Devin, V. E., Hogg, D. C. and Cohn, A. G. (2005), Protocols from perceptual observations, *Artificial Intelligence*, volume 167(1-2), pp. 103–136.
- Ng, A. Y., Harada, D. and Russell, S. J. (1999), Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping, in: *Proceedings of the International Conference on Machine Learning*

BIBLIOGRAPHY

- (ICML), pp. 278–287.
- Nienhuys-Cheng, S. H. and de Wolf, R. (1997), *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag.
- Nilsson, N. J. (1980), *Principles of Artificial Intelligence*, Tioga Pub., Palo Alto, California.
- (1986), Probabilistic Logic, *Artificial Intelligence*, volume 28, pp. 71–87.
- (1995), Eye on the Prize, *AI Magazine*, volume 16, pp. 9–16.
- (2005), Human-Level Artificial Intelligence? Be Serious!, *AI Magazine*.
- Nilsson, U. and Maluszinski, J. (1995), *Logic, Programming and Prolog*, second edition, John Wiley and Sons, Chichester, England.
- Nolfi, S. (2002), Power and Limits of Reactive Agents, *Neurocomputing*, volume 42(505), pp. 119–145.
- Nolfi, S. and Floreano, D. (2000), *Evolutionary Robotics*, The MIT Press, Cambridge, Massachusetts.
- Oates, T. and Cohen, P. R. (1996), Learning Planning Operators with Conditional and Probabilistic Effects, in: *Planning with Incomplete Information for Robot Problems: Papers from the 1996 AAAI Spring Symposium*, pp. 86–94.
- Omar, R. (1994), Artificial Intelligence through Logic?, *AI Communications*, volume 7(3/4), pp. 161–174.
- Orengo, C. A., Jones, D. T. and Thornton, J. M. (2003), *Bioinformatics: Genes, Proteins and Computers*, BIOS Scientific Publishers Ltd., Oxford, United Kingdom.
- Ormoneit, D. and Sen, S. (2002), Kernel-Based Reinforcement Learning, *Machine Learning*, volume 49, pp. 161–178.
- Ortiz, J., Garcia, A. and Borrajo, D. (2008), A Relational Learning Approach to Activity Recognition from Sensor Readings, in: *Proceedings of the 4th IEEE Intelligent Systems Conference*.
- Ortony, A., Clore, G. L. and Collins, A. (1988), *The Cognitive Structure of Emotions*, Cambridge University Press, Cambridge, MA.
- Papavassiliou, V. A. and Russell, S. J. (1999), Convergence of Reinforcement Learning with General Function Approximators, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 748–757.
- Parekh, R., Yang, J. and Honavar, V. (2000), Constructive Neural-Network Learning Algorithms for Pattern Classification, *IEEE Transactions on Neural Networks*, volume 11(2), pp. 436–451.
- Parr, R., Painter-Wakefield, C., Li, L. and Littman, M. L. (2007), Analyzing Feature Generation for Value-Function Approximation, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 737–744.
- Parr, R. and Russell, S. J. (1998), Reinforcement Learning with Hierarchies of Machines, in: *Proceedings of the Neural Information Processing Conference (NIPS)*.
- Pasula, H. M., Zettlemoyer, L. S. and Kaelbling, L. P. (2004), Learning Probabilistic Planning Rules, in: *Proceedings of the International Conference on Artificial Intelligence Planning Systems (ICAPS)*.
- (2007), Learning Symbolic Models of Stochastic Domains, *Journal of Artificial Intelligence Research (JAIR)*, volume 29, pp. 309–352.
- Pearl, J. (1988), *Probabilistic Reasoning in Intelligent Systems*, Morgan Kaufman, San Mateo, CA.
- Pednault, E. (1989), ADL: Exploring the Middle Ground between STRIPS and the Situation Calculus, in: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pp. 324–332.
- Peng, J. and Williams, R. J. (1996), Incremental Multi-Step Q-Learning, *Machine Learning*, volume 22, pp. 283–290.
- Petrick, R., Kraft, D., Mourão, K., Pugeault, N., Krüger, N. and Steedman, M. (2008), Representation and Integration: Combining Robot Control, High-Level Planning, and Action Learning, in: *Proceedings of the Sixth International Cognitive Robotics Workshop (CogRob 2008) at ECAI 2008*, pp. 32–41.
- Pfeffer, A. (2001), IBAL: A probabilistic rational programming language, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 733–740.
- Pfeifer, R. and Scheier, C. (1999), *Understanding Intelligence*, The MIT Press, Cambridge, Massachusetts.
- Piaget, J. (1950), *The Psychology of Intelligence*, Routledge & Kegan Paul, New York.
- Picard, R. W. (1997), *Affective Computing*, The MIT Press, Cambridge, Massachusetts.
- Pickett, M. and Barto, A. G. (2002), PolicyBlocks: An Algorithm for Creating Useful Macro-Actions in Reinforcement Learning, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp.

- 506–513.
- Pineau, J., Gordon, G. J. and Thrun, S. (2006), Anytime Point-Based Approximations for Large POMDPs, *Journal of Artificial Intelligence Research (JAIR)*, volume 27, pp. 335–380.
- Plotkin, G. D. (1970), A Note on Inductive Generalization, in: *Machine Intelligence*, volume 5, Edinburgh University Press, pp. 153–163.
- Poole, D. (1996), A Framework for Decision-Theoretic Planning I: Combining the Situation Calculus, Conditional Plans, Probability and Utility, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 436–445.
- (1997a), The Independent Choice Logic for Modeling Multiple Agents under Uncertainty, *Artificial Intelligence*, volume 94, pp. 7–56.
- (1997b), Probabilistic Partial Evaluation : Exploiting Rule Structure in Probabilistic Inference, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1284–1291.
- Poole, D., Mackworth, A. and Goebel, R. (1998), *Computational Intelligence: A Logical Approach*, Oxford University Press, New York.
- Porto, V. W. (1997), Neural-Evolutionary Systems, in: Fiesler, E. and Beale, R. (eds.), *Handbook of Neural Computation*, chapter D2, Institute of Physics and Oxford University Press, New York, New York.
- Potts, D. and Hengst, B. (2004a), Concurrent Discovery of Task Hierarchies, in: *AAAI Spring Symposium on Knowledge Representation and Ontology for Autonomous Systems*.
- (2004b), Discovering Multiple Levels of a Task Hierarchy Concurrently, *Robotics and Autonomous Systems*, volume 49, pp. 43–55.
- Prechelt, L. (1997), Investigation of the CasCor Family of Learning Algorithms, *Neural Networks*, volume 10(5), pp. 885–896.
- Puterman, M. L. (1994), *Markov Decision Processes—Discrete Stochastic Dynamic Programming*, John Wiley & Sons, Inc., New York, NY.
- Puterman, M. L. and Shin, M. C. (1978), Modified Policy Iteration algorithms for Discounted Markov Decision Processes, *Management Science*, volume 24, pp. 1127–1137.
- Pyeatt, L. D. and Howe, A. E. (1998), Decision Tree Function Approximation in Reinforcement Learning, *Technical Report CS-98-112*, Computer Science Department, Colorado State University.
- Quartz, S. R. (1999), The Constructivist Brain, *Trends in Cognitive Science*, volume 3(2).
- Quartz, S. R. and Sejnowski, T. J. (1997), The Neural Basis of Cognitive Development: A Constructivist Manifesto, *Journal of Brain and Behavioral Sciences*, volume 20, pp. 537–555.
- Quinlan, J. R. (1990), Learning Logical Definitions from Relational Data, *Machine Learning*, volume 5, pp. 239–266.
- Quinlan, P. T. (1998), Structural Change and Development in Real and Artificial Neural Networks, *Neural Networks*, volume 11(4), pp. 577–599.
- Rabin, S. (ed.) (2002), *AI Game Programming Wisdom*, Charles River Media, Inc., Hingham, Massachusetts.
- Ramon, J. (2002), *Clustering and Instance Based Learning in First Order Logic*, Ph.D. thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium.
- (2005a), Convergence of Reinforcement Learning using a Decision Tree Learner, in: Driessens, K., Fern, A. and van Otterlo, M. (eds.), *Proceedings of the ICML-2005 Workshop on Rich Representations for Reinforcement Learning*, pp. 51–56.
- (2005b), On the Convergence of Reinforcement Learning using a Decision Tree Learner, *Technical report*, Katholieke Universiteit Leuven, Belgium.
- Ramon, J. and Driessens, K. (2004), On the Numeric Stability of Gaussian Processes Regression for Relational Reinforcement Learning, in: *Proceedings of the Workshop on Relational Reinforcement Learning at ICML'04*.
- Ramon, J., Driessens, K. and Croonenborghs, T. (2007), Transfer Learning in Reinforcement Learning Problems Through Partial Policy Recycling, in: *Proceedings of the European Conference on Machine Learning (ECML)*.
- Rao, A. S. (1996), AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language, in: *Seventh European Workshop on Modeling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands.
- Rao, A. S. and Georgeff, M. P. (1991), Modeling rational agents within a BDI architecture, in: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pp. 473–484.

BIBLIOGRAPHY

- (1995), BDI Agents: from Theory to Practice, in: *Proceedings of the International Conference on Multi-Agent Systems (ICMAS'95)*, pp. 312–319.
- Ratitch, B. (2005), *On Characteristics of Markov Decision Processes and Reinforcement Learning in Large Domains*, Ph.D. thesis, The School of Computer Science, McGill University, Montreal.
- Ratitch, B. and Precup, D. (2004), Sparse Distributed Memories for On-Line Value-Based Reinforcement Learning, in: *Proceedings of the European Conference on Machine Learning (ECML)*, pp. 347–358.
- Ravindran, B. (2004), *An Algebraic Approach to Abstraction in Reinforcement Learning*, Ph.D. thesis, Department of Computer Science, University of Massachusetts, Amherst, MA.
- Ravindran, B. and Barto, A. G. (2001), Symmetries and Model Minimization in Markov Decision Processes, *Technical Report CMPSCI 01-43*, Department of Computer Science, University of Amherst, MA.
- (2002), Model Minimization in Hierarchical Reinforcement Learning, in: *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA)*.
- (2003a), An Algebraic Approach to Abstraction in Reinforcement Learning, in: *Proceedings of the Twelfth Yale Workshop on Adaptive and Learning Systems*, pp. 109–114.
- (2003b), SMDP Homomorphisms: An Algebraic Approach to Abstraction in Semi-Markov Decision Processes, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.
- Reed, R. D. and Marks II, R. J. (1999), *Neural Smithing*, The MIT Press, Cambridge, Massachusetts.
- Reid, M. and Ryan, M. R. K. (2000), Using ILP to Improve Planning in Hierarchical Reinforcement Learning, in: *Proceedings of the International Conference on Inductive Logic Programming (ILP)*, volume 1866 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 174–190.
- Reiter, R. (2001), *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*, The MIT Press, Cambridge, Massachusetts.
- Rennie, J. and McCallum, A. K. (1999), Using Reinforcement Learning to Spider the Web Efficiently, in: *Proceedings of the International Conference on Machine Learning (ICML)*.
- Reyes, A., Sucar, L. E., Morales, E. F. and Ibarüengoytia, P. (2003), Relational MDPs using Qualitative Proportionality Predicates: An application in Power Generation, in: *NIPS'03 workshop on: Planning for the Real World: The promises and challenges of dealing with uncertainty*.
- Reynolds, S. I. (1999), Decision Boundary Partitioning: Variable Resolution Model-Free Reinforcement Learning, *Technical Report CSRP-99-15*, University of Birmingham, School of Computer Science.
- (2000), Adaptive Resolution Model-Free Reinforcement Learning: Decision Boundary Partitioning, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 783–790.
- (2002), *Reinforcement Learning with Exploration*, Ph.D. thesis, The School of Computer Science, The University of Birmingham, UK.
- Richards, N., Moriarty, D. E. and Miikkulainen, R. (1998), Evolving Neural Networks to Play Go, *Applied Intelligence*, volume 8(1), pp. 85–96, special Issue on Evolutionary Learning.
- Richardson, M. and Domingos, P. (2006), Markov Logic Networks, *Machine Learning*, volume 62, pp. 107–136.
- Riedmiller, M., Spott, M. and Weisbrod, J. (1999), FYNESSE: A Hybrid Architecture for Self-Learning Control, in: Cloete, I. and Zurada, J. (eds.), *Knowledge-Based Neurocomputing*, The Mit Press, Cambridge, Massachusetts.
- Ring, M. B. (1994), *Continual Learning in Reinforcement Environments*, Ph.D. thesis, The University of Texas at Austin.
- (1997), CHILD: A First Step Towards Continual Learning, *Machine Learning*, volume 28, pp. 77–104.
- Rivest, F., Bengio, Y. and Kalaska, J. (2004), Brain Inspired Reinforcement Learning, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 1129–1136.
- Rivest, F. and Precup, D. (2003), Combining TD-Learning with Cascade-Correlation Networks, in: *Proceedings of the International Conference on Machine Learning (ICML)*.
- Rivest, R. L. (1987), Learning Decision Lists, *Machine Learning*, volume 2, pp. 229–246.
- Rodrigues, C., Gerard, P. and Rouveirol, C. (2008), On and Off-Policy Relational Reinforcement Learning, in: *Late-Breaking Papers of the International Conference on Inductive Logic Programming*.
- Romein, J. W. and Bal, H. E. (2003), Solving the Game of Awari using Parallel Retrograde Analysis, *IEEE Computer*, volume 36(10), pp. 26–33.
- Roncagliolo, S. and Tadepalli, P. (2004), Function Approximation in Hierarchical Relational Reinforcement

- Learning, in: *Proceedings of the Workshop on Relational Reinforcement Learning at ICML'04*.
- Roy, N. and Thrun, S. (2005), Integrating Value Functions and Policy Search for Continuous Markov Decision Processes, in: *Proceedings of the Neural Information Processing Conference (NIPS)*.
- Rummery, G. A. (1995), *Problem Solving with Reinforcement Learning*, Ph.D. thesis, Cambridge University, Engineering Department, Cambridge, England.
- Rummery, G. A. and Niranjana, M. (1994), On-Line Q-Learning using Connectionist Systems, *Technical Report CUED/F-INFENG/TR 166*, Cambridge University, Engineering Department.
- Russell, S. J. (1997), Rationality and Intelligence, *Artificial Intelligence*, volume 94, pp. 57–77.
- Russell, S. J. and Norvig, P. (2003), *Artificial Intelligence: a Modern Approach*, Prentice Hall, New Jersey, 2nd edition.
- Ryan, M. R. K. (2002), Using Abstract Models of Behaviors to Automatically Generate Reinforcement Learning Hierarchies, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 522–529.
- (2004a), Hierarchical Decision Making, in: Si *et al.* (2004).
- (2004b), *Hierarchical Reinforcement Learning: A Hybrid Approach*, Ph.D. thesis, University of NSW, School of Computer Science and Engineering, Sidney, Australia.
- Rylatt, R. M. and Czarnecki, C. A. (2000), Embedding Connectionist Autonomous Agents in Time: The 'Road Sign Problem', *Neural Processing Letters*, volume 12, pp. 145–158.
- Saad, E. (2008), A Logical Framework to Reinforcement Learning using Hybrid Probabilistic Logic Programs, in: *Proceedings of the International Conference on Scalable Uncertainty Management (SUM)*, volume 5291 of *Lecture Notes in Artificial Intelligence*, pp. 341–355.
- Sablon, G. and Bruynooghe, M. (1994), Using the Event Calculus to Integrate Planning and Learning in an Intelligent Autonomous Agent, in: Bäckström, C. and Sandewall, E. (eds.), *Current Trends in AI Planning*, IOS Press, pp. 254–265.
- Safaei, J. and Ghassem-Sani, G. (2007), Incremental Learning of Planning Operators in Stochastic Domains, in: *Proceedings of the International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pp. 644–655.
- Saitta, L. (1996), Representation Change in Machine Learning, *Communications of the ACM*, volume 9, pp. 14–20.
- Sallans, B. (2002), *Reinforcement Learning for Factored Markov Decision Processes*, Ph.D. thesis, Department of Computer Science, University of Toronto, Toronto.
- Sallans, B. and Hinton, G. E. (2004), Reinforcement Learning with Factored States and Actions, *Journal of Artificial Intelligence Research (JAIR)*, volume 5, pp. 1063–1088.
- Samuel, A. L. (1959), Some Studies in Machine Learning using the Game of Checkers, *IBM Journal of Research and Development*, volume 3, pp. 210–229.
- Sanghai, S., Domingos, P. and Weld, D. S. (2003), Dynamic Probabilistic Relational Models, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.
- Sanner, S. (2005), Simultaneous Learning of Structure and Value in Relational Reinforcement Learning, in: Driessens, K., Fern, A. and van Otterlo, M. (eds.), *Proceedings of the ICML-2005 Workshop on Rich Representations for Reinforcement Learning*.
- (2006a), Online Feature Discovery in Relational Reinforcement Learning, in: *Proceedings of the ICML-06 Workshop on Open Problems in Statistical Relational Learning*.
- (2006b), Thesis Summary: First-Order Decision-Theoretic Planning, abstract for the ICAPS-2006 Doctoral Consortium.
- (2008a), *First-Order Decision-Theoretic Planning in Structured Relational Environments*, Ph.D. thesis, Department of Computer Science, University of Toronto, Canada.
- (2008b), How to Spice up your Planning under Uncertainty Research Life, in: *Proceedings of the ICAPS-2008 Workshop on A Reality Check for Planning and Scheduling Under Uncertainty*.
- Sanner, S. and Boutilier, C. (2005), Approximate Linear Programming for First-Order MDPs, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*.
- (2006), Practical Linear Value-approximation Techniques for First-Order MDPs, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*.
- (2007), Approximate Solution Techniques for Factored First-Order MDPs, in: *Proceedings of the Inter-*

BIBLIOGRAPHY

- national Conference on Artificial Intelligence Planning Systems (ICAPS).*
- Sanner, S. and Kersting, K. (2007), Symbolic Dynamic Programming, in: Sammut, C. (ed.), *Encyclopedia of Machine Learning*, Springer-Verlag.
- Santamaria, J. C., Sutton, R. S. and Ram, A. (1997), Experiments with Reinforcement Learning in Problems with Continuous State and Action Spaces, *Adaptive Behavior*, volume 6(2).
- Santos, M. V. (2000), Specifying and Reasoning about actions in open-worlds using Transaction Logic, in: *The Workshop on Cognitive Robotics at ECAI-00.*
- Sato, T. (1995), A Statistical Learning Method for Logic Programs with Distribution Semantics, in: *Proceedings of the Twelfth International Conference on Logic Programming (ICLP'95)*, pp. 715–729.
- (2001), Parameterized Logic Programs where Computing meets Learning, in: *Proceedings of the International Symposium on Functional and Logic Programming (FLOPS)*, volume 2024 of *Lecture Notes in Computer Science*, pp. 40–60.
- Scarselli, F. and Tsoi, A. C. (1998), Universal Approximation Using Feedforward Neural Networks: A Survey of Some Existing Methods, and Some New Results, *Neural Networks*, volume 11(1), pp. 15–37.
- Schaeffer, J. and Plaatt, A. (1997), Kasparov versus Deep Blue: The Re-Match, *International Computer Chess Association Journal*, volume 20(2), pp. 95–101.
- Schmid, U. (2001), Inductive Synthesis of Functional Programs: Learning Domain-Specific Control Rules and Abstraction Schemes, habilitationsschrift, Fakultät IV, Elektrotechnik und Informatik, Technische Universität Berlin, Germany.
- Schmidhuber, J. H. (2000), Evolutionary Computation versus Reinforcement Learning, in: *Annual Conference of the IEEE Industrial Electronics Society (IECON)*, pp. 2992–2997.
- Schölkopf, B. and Smola, A. J. (2002), *Learning with Kernels*, The MIT Press, Cambridge, Massachusetts.
- Schuermans, D. and Patrascu, R. (2001), Direct Value Approximation for Factored MDPs, in: *Proceedings of the Neural Information Processing Conference (NIPS).*
- Schwartz, A. (1993), A Reinforcement Learning Method for Maximizing Undiscounted Rewards, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 298–305.
- Schwind, C. (1999), Causality in Action Theories, *Electronic Transactions on Artificial Intelligence*, volume 3, pp. 27–50.
- Scott, P. D. and Markovitch, S. (1991), Representation Generation in An Exploratory Learning System, in: Fisher, D., Pazzani, M. and Langley, P. (eds.), *Concept Formation: Knowledge and Experience in Unsupervised Learning*, chapter 14, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Shahaf, D. and Amir, E. (2006), Learning Partially Observable Action Schemas, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI).*
- Shapiro, D. and Langley, P. (2002), Separating Skills from Preference, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 570–577.
- Shen, W. M. (1993), Discovery as Autonomous Learning from the Environment, *Machine Learning*, volume 12, pp. 143–165.
- Shultz, T. R. (2003), *Computational Developmental Psychology*, The MIT Press, Cambridge, Massachusetts.
- Shultz, T. R. and Rivest, F. (2001), Knowledge-Based Cascade-Correlation: Using Knowledge to Speed Up Learning, *Connection Science*, volume 13, pp. 1–30.
- Si, J., Barto, A. G., Powell, W. B. and Wunsch, D. (eds.) (2004), *Handbook of Learning and Approximate Dynamic Programming*, Wiley-IEEE Press, Piscataway, NJ.
- Simari, G. I. and Parsons, S. (2006), On The Relationship between MDPs and the BDI Architecture, in: *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pp. 1041–1048.
- Simsek, O. and Barto, A. G. (2004), Using Relative Novelty to Identify Useful Temporal Abstractions in Reinforcement Learning, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 751–758.
- Simsek, O., Wolfe, A. P. and Barto, A. G. (2005), Identifying Useful Subgoals in Reinforcement Learning by Local Graph Partitioning, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 817–824.
- Singh, S. (1992), Reinforcement Learning with a Hierarchy of Abstract Models, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 202–207.

- Singh, S., Barto, A. G. and Chentanez, N. (2005), Intrinsically Motivated Reinforcement Learning, in: *Proceedings of the Neural Information Processing Conference (NIPS)*.
- Singh, S. and Cohn, D. (1998), How to Dynamically Merge Markov Decision Processes, in: *Proceedings of the Neural Information Processing Conference (NIPS)*.
- Singh, S., Jaakkola, T. and Jordan, M. I. (1995), Reinforcement Learning with Soft State Aggregation, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 361–368.
- Singh, S., James, M. R. and Rudary, M. R. (2004), Predictive State Representations: A New Theory for Modeling Dynamic Systems, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 512–519.
- Skvortsova, O. (2003), *Towards Automated Symbolic Dynamic Programming*, Master’s thesis, TU Dresden.
- (2006a), Lifted First-Order Decision-Theoretic Planning, in: *Tenth Russian Conference on Artificial Intelligence (CAI’2006)*.
- (2006b), A new Context-Based θ -Subsumption Algorithm, in: *Proceedings of the International Conference on Inductive Logic Programming (ILP)*, short paper.
- (2006c), Say No to Grounding: An Inference Algorithm for First-Order MDPs, abstract for the ICAPS-2006 Doctoral Consortium.
- Slaney, J. (1995), Generating Random States of Blocks Worlds, *Technical Report TR-ARP-18-95*, Research School of Information Sciences and Engineering and Centre for Information Science Research, Australian National University.
- Slaney, J. and Thiébaux, S. (1994), Adventures in Blocks World, *Technical Report TR-ARP-7-94*, Research School of Information Sciences and Engineering and Centre for Information Science Research, Australian National University.
- (2001), Blocks World Revisited, *Artificial Intelligence*, volume 125, pp. 119–153.
- Smart, W. D. (2004), Explicit Manifold Representations for Value-Function Approximation in Reinforcement Learning, in: *Proceedings of AI-MATH 2004*.
- Smart, W. D. and Kaelbling, L. P. (2000), Practical Reinforcement Learning in Continuous Spaces, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 903–910.
- Smith, T. and Simmons, R. (2005), Point-Based POMDP Algorithms: Improved Analysis and Implementation, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 542–547.
- Smolensky, P. (1989), Neural Connections, Mental Computation, chapter Connectionist Modeling: Neural Computation / Mental Connections, Bradford/The MIT Press, Cambridge, Massachusetts, reprinted in *Mind Design II: Philosophy, Psychology, Artificial Intelligence* (1997) by J. Haugeland (ed.), MIT Press, Cambridge, Massachusetts.
- Song, Z. W. and Chen, X. P. (2006), Unique State and Automatic Action Abstracting Based on Logical MDPs with Negation, in: *Proceedings of the International Conference on Natural Computation (ICNC)*, volume 4222 of *Lecture Notes in Computer Science*, pp. 928–937.
- (2007), States Evolution in $\Theta(\lambda)$ -Learning based on Logical MDPs with Negation, in: *IEEE International Conference on Systems, Man and Cybernetics*, pp. 1624–1629.
- (2008), Agent Learning in Relational Domains based on Logical MDPs with Negation, *Journal of Computers*, volume 3(9), pp. 29–38.
- Soutchanski, M. (2001), An On-Line Decision-Theoretic Golog Interpreter, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.
- Sowa, J. F. (1999), *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Thomson Learning, Stamford, Connecticut.
- Spaan, M. T. J. (2006), *Approximate Planning under Uncertainty in Partially Observable Environments*, Ph.D. thesis, Universiteit van Amsterdam.
- Spruit, P. A. (1994), *Logics of Database Updates*, Ph.D. thesis, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, the Netherlands.
- Srinivasan, A. (1999), A Study of Two Probabilistic Methods for Searching Large Spaces with ILP, *Data Mining and Knowledge Discovery*, volume 3(1), pp. 95–123.
- St-Aubin, R., Hoey, J. and Boutilier, C. (2001), APRICODD: Approximate Policy Construction using Decision Diagrams, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 1089–1095.
- Stanley, K. and Miikkulainen, R. (2001), Evolving neural networks through augmenting topologies, *Technical*

BIBLIOGRAPHY

- Report AI-01-290*, Department of Computer Sciences, The University of Texas at Austin.
- Steinkraus, K. and Kaelbling, L. P. (2004), Combining Dynamic Abstractions in Large MDPs, *Technical Report AIM-2004-023*, MIT Computer Science and Artificial Intelligence Laboratory (CSAIL).
- Sterling, L. and Shapiro, E. (1994), *The Art of Prolog*, The MIT Press, Cambridge, Massachusetts, 2nd edition.
- Stolzmann, W. (2000), An Introduction to Anticipatory Classifier Systems, in: Lanzi, P., Stolzmann, W. and Wilson, S. (eds.), *Learning Classifier Systems: From Foundations to Applications*, volume 1813 of *Lecture Notes in Artificial Intelligence*, pp. 175–194.
- Stone, P. (2007), Learning and Multiagent Reasoning for Autonomous Agents, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Computers and Thought Award Paper.
- Stone, P. and Veloso, M. (2000), Multiagent Systems: A Survey from a Machine Learning Perspective, *Autonomous Robotics*, volume 8(3).
- Stracuzzi, D. J. and Asgharbeygi, N. (2006), Transfer of Knowledge Structures with Relational Temporal Difference Learning, in: *Proceedings of the ICML'06 Workshop on Structural Knowledge Transfer for Machine Learning*.
- Strehl, A., Diuk, C. and Littman, M. L. (2007), Efficient Structure Learning in Factored-state MDPs, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.
- Struyf, J. (2004), *Techniques for Improving the Efficiency of Inductive Logic Programming in the Context of Data Mining*, Ph.D. thesis, Department of Computer Science, Catholic University of Leuven, Belgium.
- Sun, R. (1998), Supplementing Neural Reinforcement Learning with Symbolic Methods, in: *Hybrid Neural Systems*, pp. 333–347.
- Sun, R. and Peterson, T. (1998), Autonomous Learning of Sequential Tasks: Experiments and Analysis, *IEEE Transactions on Neural Networks*, volume 9(6), pp. 1217–1234.
- Sutton, R. S. (1988), Learning to Predict by the Methods of Temporal Differences, *Machine Learning*, volume 3, pp. 9–44.
- (1990), Integrated Architectures for Learning, Planning, and Reacting based on Approximating Dynamic Programming, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 216–224.
- (1991a), DYNA, an Integrated Architecture for Learning, Planning and Reacting, in: *Working Notes of the AAAI Spring Symposium on Integrated Intelligent Architectures*, pp. 151–155.
- (1991b), Reinforcement Learning Architectures for Animats, in: *From Animals to Animats: Proceedings of The International Conference on Simulation of Adaptive Behavior (SAB)*, pp. 288–296.
- (1996), Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 1038–1044.
- (1997), On the Significance of Markov Decision Processes, in: *Proceedings of the International Conference on Artificial Neural Networks*, pp. 273–282.
- (1999), Reinforcement Learning: Past, Present and Future, in: *Proceedings of the Second Asia Pacific Conference on Simulated Evolution and Learning (SEAL'98)*, volume 1585 of *Lecture Notes in Computer Science*, pp. 195–197.
- (2007), The PEAK Project, unpublished.
- Sutton, R. S. and Barto, A. G. (1998), *Reinforcement Learning: an Introduction*, The MIT Press, Cambridge, Massachusetts.
- Sutton, R. S., McAllester, D. A., Singh, S. and Mansour, Y. (2000), Policy Gradient Methods for Reinforcement Learning with Function Approximation, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 1057–1063.
- Sutton, R. S., Precup, D. and Singh, S. (1999), Between MDPs and semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning, *Artificial Intelligence*, volume 112(1–2), pp. 181–211.
- Sutton, R. S., Rafols, E. J. and Koop, A. (2005), Temporal Abstraction in Temporal-Difference Networks, in: *Proceedings of the Neural Information Processing Conference (NIPS)*.
- Sutton, R. S. and Tanner, B. (2005), Temporal-Difference Networks, in: *Proceedings of the Neural Information Processing Conference (NIPS)*.
- Sutton, R. S. and Whitehead, S. D. (1993), Online Learning with Random Representations, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 314–321.
- Szepesvari, C. and Smart, W. D. (2004), Interpolation-based Q-learning, in: *Proceedings of the International*

- Conference on Machine Learning (ICML)*, pp. 791–798.
- Tadepalli, P. and Dietterich, T. G. (1997), Hierarchical Explanation-Based Reinforcement Learning, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 358–366.
- Tadepalli, P., Givan, R. and Driessens, K. (2004), Relational Reinforcement Learning: An Overview, in: *Proceedings of the Workshop on Relational Reinforcement Learning at ICML'04*.
- Tan, M. (1993), Multi-Agent Reinforcement Learning: Independent Vs. Cooperative Agents, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 330–337.
- Tash, J. and Russell, S. J. (1994), Control Strategies for a Stochastic Planner, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 1079–1085.
- Taskar, B., Abbeel, P. and Koller, D. (2002), Discriminative Probabilistic Models for Relational Data, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*.
- Taylor, M. E., Whiteson, S. and Stone, P. (2006), Comparing Evolutionary and Temporal Difference Methods in a Reinforcement Learning Domain, in: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1321–1328.
- Teichteil-Koenigsbuch, F. (2008), Extending PPDDL1.0 to Model Hybrid Markov Decision Processes, in: *Proceedings of the ICAPS-2008 Workshop on A Reality Check for Planning and Scheduling Under Uncertainty*.
- Tesauro, G. J. (1994), TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play, *Neural Computation*, volume 6, pp. 215–219.
- Theocharous, G. (2002), *Hierarchical Learning and Planning in Partially Observable Markov decision Processes*, Ph.D. thesis, Department of Computer Science, Michigan State University.
- Thielscher, M. (1998), Introduction to the Fluent Calculus, *Electronic Transactions on Artificial Intelligence*, volume 2(3–4), pp. 179–192.
- (1999), From Situation Calculus to Fluent Calculus: State Update Axioms as a Solution to the Inferential Frame Problem, *Artificial Intelligence*, volume 111, pp. 277–299.
- (2005), FLUX: A Logic Programming Method for Reasoning Agents, *Theory and Practice of Logic Programming*, volume 5, pp. 533–565.
- Thon, I., Landwehr, N. and De Raedt, L. (2008), A Simple Model for Sequences of Relational State Descriptions, in: *Proceedings of the European Conference on Machine Learning (ECML)*, volume 5212 of *Lecture Notes in Computer Science*, Springer, pp. 506–521.
- Thorndike, E. L. (1911), *Animal Intelligence*, Hafner, Darien, CT.
- Thornton, C. (1996), Parity: The Problem that Won't Go Away, in: *Advances in Artificial Intelligence*, volume 1081 of *Lecture Notes in Computer Science*, pp. 362–374.
- (1999), What do Constructive Learners Really Learn?, *Artificial Intelligence Review*, volume 13(4), pp. 249–257.
- (2000), *Truth from Trash: How Learning makes Sense*, The MIT Press, Cambridge, Massachusetts.
- Thornton, S. (2002), *Growing Minds: An Introduction to Cognitive Development*, Palgrave MacMillan, New York.
- Thrun, S. (1996), *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*, Kluwer Academic Publishers.
- Thrun, S. and Schwartz, A. (1993), Issues in Using Function Approximation for Reinforcement Learning, in: *Proceedings of the 1993 Connectionist Models Summer School*.
- (1995), Finding Structure in Reinforcement Learning, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 385–392.
- Torrey, L., Shavlik, J., Natarajan, S., Kuppili, P. and Walker, T. (2008), Transfer in Reinforcement Learning via Markov Logic Networks, in: *Proceedings of the AAAI-2008 Workshop on Transfer Learning for Complex Tasks*.
- Torrey, L., Shavlik, J., Walker, T. and Maclin, R. (2006), Relational Skill Transfer via Advice Taking, in: *Proceedings of the ICML-06 Workshop on Structural Knowledge Transfer for Machine Learning*.
- (2007), Relational Macros for Transfer in Reinforcement Learning, in: *Proceedings of the International Conference on Inductive Logic Programming (ILP)*.
- Towell, G. G. and Shavlik, J. (1994), Knowledge-based Artificial Neural Networks, *Artificial Intelligence*, volume 70(1–2), pp. 119–165.
- Tran, S. D. and Davis, L. S. (2008), Event Modeling and Recognition Using Markov Logic Networks, in:

BIBLIOGRAPHY

- European Conference on Computer Vision*, pp. II: 610–623.
- Tsitsiklis, J. and van Roy, B. (1997), An Analysis of Temporal-Difference Learning with Function Approximation, *IEEE Transactions on Automatic Control*, volume 42, pp. 674–690.
- Tsoi, A. C. (1998), Recurrent Neural Network Architectures: An Overview, *Lecture Notes in Computer Science*, volume 1387, pp. 1–26.
- Turaga, P., Chellappa, R., Subrahmanian, V. S. and Udrea, O. (2008), Machine Recognition of Human Activities: A Survey, *IEEE Transactions on Circuits, Systems and Video Technology*, volume 18(11), special issue on Event Analysis.
- Tuyls, K., Croonenborghs, T., Ramon, J., Goetschalckx, R. and Bruynooghe, M. (2005), Multi-Agent Relational Reinforcement Learning, in: *Proceedings of the LAMAS Workshop at AAMAS-2005*, position paper.
- Tversky, A. and Kahneman, D. (1981), The Framing of Decisions and the Psychology of Choice, *Science*, volume 211(4481), pp. 453–458.
- Utgoff, P. E. (1996), Feature Function Learning for Value Function Approximation, *Technical Report UM-CS-1996-009*, University of Massachusetts, Amherst, Computer Science.
- (1997), Decision Tree Induction based on Efficient Tree Restructuring, *Machine Learning*, volume 29(1), pp. 5–44.
- Utgoff, P. E. and Precup, D. (1997), Constructive Function Approximation, *Technical report*, University of Massachusetts at Amherst, Amherst, MA.
- (1998), Constructive Function Approximation, in: Liu, H. and Motoda, H. (eds.), *Feature Extraction, Construction and Selection: A Data Mining Perspective*, volume 453 of *The Kluwer International Series in Engineering and Computer Science*, chapter 14, Kluwer Academic Publishers.
- Utgoff, P. E. and Stracuzzi, D. J. (2002), Many-Layered Learning, *Neural Computation*, volume 14(10), pp. 2497–2529.
- Uther, W. T. B. and Veloso, M. (1998), Tree Based Discretization for Continuous State Space Reinforcement Learning, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.
- Vamplew, P. and Ollington, R. (2005a), Global versus Local Constructive Function Approximation for On-Line Reinforcement Learning, in: *Proceedings of 18th Australian Joint Conference on Artificial Intelligence (AI'05)*.
- (2005b), On-Line Reinforcement Learning using Cascade Constructive Neural Networks, in: *Proceedings of 9th International Conference on Knowledge-Based Intelligent Information & Engineering Systems (KES)*.
- van der Meulen, P. G. M., Schipper, H., Bazen, A. M. and Gerez, S. H. (2001), PMDGP: A Distributed Object-Oriented Genetic Programming Environment, in: *Proceedings ASCI Conference 2001*.
- van Harmelen, F. and Bundy, A. (1988), Explanation-Based Generalization = Partial Evaluation, *Artificial Intelligence*, volume 36, pp. 401–412.
- van Laer, W. (2002), *From Propositional to First Order Logic in Machine Learning and Data Mining - Induction of first order rules with ICL*, Ph.D. thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium.
- van Laer, W. and De Raedt, L. (2001a), How to Upgrade Propositional Learners to First Order Logic: A Case Study, in: *Proceedings of ACA'99*, volume 2049 of *Lecture Notes in Artificial Intelligence*, pp. 102–126.
- (2001b), How to Upgrade Propositional Learners to First Order Logic: A Case Study, *Technical Report CW-288*, Department of Computer Science, Catholic University of Leuven.
- van Otterlo, M. (2002), Relational Representations in Reinforcement Learning: Review and Open Problems, in: de Jong, E. D. and Oates, T. (eds.), *Proceedings of the ICML'02 Workshop on Development of Representations*.
- (2003), Efficient Reinforcement Learning using Relational Aggregation, in: *Proceedings of the Sixth European Workshop on Reinforcement Learning, Nancy, France (EWRL-6)*.
- (2004a), Reinforcement Learning for Relational MDPs, in: Nowé, A., Lenaerts, T. and Steenhaut, K. (eds.), *Machine Learning Conference of Belgium and the Netherlands (BeNeLearn'04)*, pp. 138–145.
- (2004b), Thesis Review of: Relational Reinforcement Learning by K. Driessens, newsletter of the Dutch-Belgium Newsletter on Artificial Intelligence.
- (2005), A Survey of Reinforcement Learning in Relational Domains, *Technical Report TR-CTIT-05-31*, CTIT, University of Twente, Enschede, The Netherlands.
- (2008a), *The Logic of Adaptive Behavior: Knowledge Representation and Algorithms the Markov Decision*

BIBLIOGRAPHY

- Process Framework in First-Order Domains*, Ph.D. thesis, Department of Computer Science, University of Twente, Enschede, The Netherlands, may, 512pp.
- (2008b), Relational Reinforcement Learning: Lifting Propositional Algorithms and Size Matters, *ÖGAI journal*, volume 27(1), pp. 7–14, ISSN 0254–4326.
- van Otterlo, M. and Kersting, K. (2004), Challenges for Relational Reinforcement Learning, in: *Proceedings of the Workshop on Relational Reinforcement Learning at ICML'04*.
- van Otterlo, M., Kersting, K. and De Raedt, L. (2004), Bellman goes Relational (extended abstract), in: *Proceedings of the Belgium-Netherlands Artificial Intelligence Conference (BNAIC)*.
- van Otterlo, M., Wiering, M. A., Dastani, M. and Meyer, J. J. (2003), A Characterization of Sapient Agents, in: *Proceedings of the International Conference on the Integration of Knowledge Intensive Multi-Agent Systems KIMAS'03*.
- (2007), A Characterization of Sapient Agents, in: Mayorga, R. V. and Perlovsky, L. I. (eds.), *Toward Computational Sapience: Principles and Systems*, chapter 9, Springer.
- Vere, S. A. (1977), Induction of Relational Productions in the Presence of Background Information, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 349–355.
- (1978), Inductive Learning of Relational Productions, in: Waterman, D. A. and Hayes-Roth, F. (eds.), *Pattern Directed Inference Systems*, Academic Press, New York.
- Vernon, D., Metta, G. and Sandini, G. (2007), A Survey of Artificial Cognitive Systems: Implications for the Autonomous Development of Mental Capabilities in Computational Agents, *Evolutionary Computation, IEEE Transactions on*, volume 11(2), pp. 151–180.
- Vollbrecht, H. (1999), kd-Q-Learning with Hierarchic Generalization in Action and State Space, *Technical Report 1999/8*, Dept. of Neural Information Processing, University of Ulm, Germany.
- Waldinger, R. (1975), Learning Structural Descriptions from Examples, in: Winston, P. (ed.), *The Psychology of Computer Vision*, McGraw-Hill, New York.
- (1977), Achieving Several Goals Simultaneously, in: Elcock, E. W. and Michie, D. (eds.), *Machine Intelligence 8*, Halstead and Wiley, New York.
- Walker, T., Shavlik, J. and Maclin, R. (2004), Relational Reinforcement Learning via Sampling the Space of First-Order Conjunctive Features, in: *Proceedings of the Workshop on Relational Reinforcement Learning at ICML'04*.
- Walker, T., Torrey, L., Shavlik, J. and Maclin, R. (2007), Building Relational World Models for Reinforcement Learning, in: *Proceedings of the International Conference on Inductive Logic Programming (ILP)*.
- Walsh, T. J., Li, L. and Littman, M. L. (2006), Transferring State Abstractions between MDPs, in: *ICML-06 Workshop on Structural Knowledge Transfer for Machine Learning*.
- Walsh, T. J. and Littman, M. L. (2008), Efficient Learning of Action Schemas and Web-Service Descriptions, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.
- Wang, C. (2007), *First-Order Markov Decision Processes*, Ph.D. thesis, Department of Computer Science, Tufts University, U.S.A.
- Wang, C., Joshi, S. and Khardon, R. (2007), First Order Decision Diagrams for Relational MDPs, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.
- (2008a), First Order Decision Diagrams for Relational MDPs, *Journal of Artificial Intelligence Research (JAIR)*, volume 31, pp. 431–472.
- Wang, C. and Khardon, R. (2007), Policy Iteration for Relational MDPs, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*.
- Wang, C. and Schmolze, J. (2005), Planning with POMDPs using a Compact, Logic-Based Representation, in: *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*.
- Wang, W., Gao, Y., Chen, X. and Ge, S. (2008b), Reinforcement Learning with Markov Logic Networks, in: *Proceedings of the Mexican Conference on Artificial Intelligence*, volume 5317 of *Lecture Notes in Computer Science*, pp. 230–242.
- Wang, X. (1995), Learning by Observation and Practice: An Incremental Approach for Planning Operator Acquisition, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 549–557.
- Wang, X. and Dietterich, T. G. (1999), Efficient Value Function Approximation Using Regression Trees, *Technical report*, Department of Computer Science, Oregon State University.
- (2003), Model-Based Policy Gradient Reinforcement Learning, in: *Proceedings of the International*

BIBLIOGRAPHY

Conference on Machine Learning (ICML).

- Wang, Y. and Laird, J. E. (2007), The Importance of Action History in Decision Making and Reinforcement Learning, in: *Proceedings of the International Conference on Cognitive Modeling*.
- Watkins, C. J. C. H. (1989), *Learning from Delayed Rewards*, Ph.D. thesis, King's College, Cambridge, England.
- Watkins, C. J. C. H. and Dayan, P. (1992), Q-Learning, *Machine Learning*, volume 8(3/4), special Issue on Reinforcement Learning.
- Weiss, G. (1999), *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, The MIT Press, Cambridge, Massachusetts.
- Weld, D. S. (1999), Recent Advances in AI Planning, *AI Magazine*, volume 20(2), pp. 93–123.
- Wermter, S. and Sun, R. (2000), *Hybrid neural systems*, volume 1778, Springer-Verlag Inc., New York, NY, USA.
- Westermann, G. (2000), *Constructivist Neural Network Models of Cognitive Development*, Ph.D. thesis, The University of Edinburgh, Great Britain.
- Whitehead, S. D. and Ballard, D. (1991), Learning to Perceive and Act, *Machine Learning*, volume 7, pp. 45–83.
- Widmer, G. and Kubat, M. (1996), Learning in the Presence of Concept Drift and Hidden Contexts, *Machine Learning*, volume 23, pp. 69–101.
- Wiering, M. A. (1995), *TD-Learning of Game Evaluation Functions with Hierarchical Neural Architectures*, Master's thesis, University of Amsterdam, The Netherlands.
- (1999), *Explorations in Efficient Reinforcement Learning*, Ph.D. thesis, Faculteit der Wiskunde, Informatica, Natuurkunde en Sterrenkunde, Universiteit van Amsterdam.
- (2000), Multi-Agent Reinforcement Learning for Traffic Light Control, in: *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 1151–1158.
- (2002), Model-Based Reinforcement Learning in Dynamic Environments, *Technical Report UU-CS-2002-029*, Institute of Information and Computing Sciences, University of Utrecht, The Netherlands.
- (2004), Convergence and Divergence in Standard and Averaging Reinforcement Learning, in: *Proceedings of the European Conference on Machine Learning (ECML)*, pp. 477–488.
- (2005), QV(λ)-Learning: A New On-policy Reinforcement Learning Algorithm, in: *Proceedings of the 7th European Workshop on Reinforcement Learning*.
- Wiering, M. A., Kröse, B. and Groen, F. (2000), Learning in Multi-Agent Systems, unpublished.
- Wiering, M. A. and Schmidhuber, J. H. (1997), HQ-learning, *Adaptive Behavior*, volume 6(2), pp. 219–246.
- (1998a), Efficient Model-Based Exploration, in: *From Animals to Animats: Proceedings of The International Conference on Simulation of Adaptive Behavior (SAB)*, pp. 223–228.
- (1998b), Fast Online Q(λ), *Machine Learning*, volume 33(1), pp. 105–115.
- Williams, R. J. (1988), Toward a Theory of Reinforcement Learning Connectionist Systems, *Technical Report NU-CCS-88-3*, Northeastern University, Boston, MA.
- (1992), Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning, *Machine Learning*, volume 8, pp. 229–256.
- Wilson, A. (2004), Generalization in Relational Reinforcement Learning (Abstract), in: *Proceedings of the ICML Workshop on Relational Reinforcement Learning*.
- Wingate, D. and Seppi, K. D. (2005), Prioritization Methods for Accelerating MDP Solvers, *Journal of Machine Learning Research (JMLR)*, volume 6, pp. 851–881.
- Wingate, D., Soni, V., Wolfe, B. and Singh, S. (2007), Relational Knowledge with Predictive State Representations, in: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.
- Winston, W. L. (1991), *Operations Research Applications and Algorithms*, Thomson Information/Publishing Group, Boston, 2nd edition.
- Witkowski, C. M. (1997), *Schemes for Learning and Behavior: A New Expectancy Model*, Ph.D. thesis, Department of Computer Science, Queen Mary Westfield College, University of London, Great Britain.
- (2007), An Action-Selection Calculus, *Adaptive Behavior*, volume 15, pp. 73–97.
- Witten, I. H. (1977), An Adaptive Optimal Controller for Discrete-Time Markov Environments, *Information and Control*, volume 34, pp. 286–295.
- Wolpert, D. H. (1995), The Relationship Between PAC, the Statistical Physics Framework, the Bayesian Framework, and the VC Framework, in: Wolpert, D. H. (ed.), *The Mathematics of Generalization*,

BIBLIOGRAPHY

- Addison-Wesley, Reading, MA, pp. 117–214.
- Wooldridge, M. (2002), *An introduction to MultiAgent Systems*, John Wiley & Sons Ltd., West Sussex, England.
- Wooldridge, M. and Jennings, N. R. (1995), Intelligent agents: Theory and Practice, *The Knowledge Engineering Review*, volume 10(2), pp. 115–152.
- Wrobel, S. (2001), Inductive Logic Programming for Knowledge Discovery in Databases, in: Džeroski and Lavrac (2001b), chapter 4, pp. 74–101.
- Wu, J. H. (2007), Discovering and Applying Domain Features in Probabilistic Planning, in: *International Conference on Automated Planning and Scheduling, Doctoral Consortium*.
- Wu, J. H. and Givan, R. (2004), Feature-Discovering Approximate Value Methods, *Technical Report TR-ECE-04-06*, School of Electrical and Computer Engineering, Purdue University.
- (2005), Feature-Discovering Approximate Value Iteration Methods, in: *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA)*, pp. 321–331.
- (2007a), Discovering Relational Domain Features for Probabilistic Planning, in: *Proceedings of the International Conference on Artificial Intelligence Planning Systems (ICAPS)*.
- (2007b), Incorporating Search Techniques in Feature-Discovering Planning, in: *Workshop on Artificial Intelligence Planning and Learning at the International Conference on Automated Planning Systems*.
- Wu, K., Yang, Q. and Jiang, Y. (2005), ARMS: Action-Relation Modelling System for Learning Action Models, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.
- Wynne-Jones, M. (1991), Node Splitting: A constructive Algorithm for feed-forward neural networks, in: *Proceedings of the Neural Information Processing Conference (NIPS)*, pp. 1072–1079.
- Yang, Q., Wu, K. and Jiang, Y. (2005), Learning Action Models from Plan Examples with Incomplete Knowledge, in: *Proceedings of the International Conference on Artificial Intelligence Planning Systems (ICAPS)*.
- (2007), Learning Action Models from Plan Examples Using Weighted MAX-SAT, *Artificial Intelligence*, volume 171(2–3), pp. 107–143.
- Yao, X. (1999), Evolving Artificial Neural Networks, *Proceedings of the IEEE*, volume 87(9), pp. 1423–1447.
- Yee, R. C., Saxena, S., Utgoff, P. E. and Barto, A. G. (1990), Explaining Temporal Differences to Create Useful Concepts for Evaluating States, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 882–888.
- Yoon, S. W. (2006), *Learning Control Knowledge for AI Planning Domains*, Ph.D. thesis, Purdue University, Indiana, United States.
- Yoon, S. W., Fern, A. and Givan, R. (2002), Inductive Policy Selection for First-Order MDPs, in: *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*.
- (2004), Learning Reactive Policies for Probabilistic Planning Domains, ICAPS’04 planning competition, probabilistic track.
- (2005), Learning Measures of Progress for Planning Domains, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.
- (2006a), Discrepancy Search with Reactive Policies for Planning, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.
- (2006b), Learning Heuristic Functions from Relaxed Plans, in: *Proceedings of the International Conference on Artificial Intelligence Planning Systems (ICAPS)*.
- Younes, H. L. S. and Littman, M. L. (2004), PPDDL 1.0: An Extension to PDDL for Expressing Planning Domains with Probabilistic Effects, *Technical Report CMU-CS-04-167*, School of Computer Science, Carnegie Mellon University, Pittsburgh.
- Younes, H. L. S., Littman, M. L., Weissman, D. and Asmuth, J. (2005), The First Probabilistic Track of the International Planning Competition, *Journal of Artificial Intelligence Research (JAIR)*, volume 24, pp. 851–887.
- Zettlemoyer, L. S., Pasula, H. M. and Kaelbling, L. P. (2005), Learning Planning Rules in Noisy Stochastic Worlds, in: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.
- Zhang, C. and Baras, J. (2001), A New Adaptive Aggregation Algorithm for Infinite Horizon Dynamic Programming, *Technical Report CSHCN TR 2001-5 (ISR TR 2001-12)*, The Center for Satellite and Hybrid Communication Networks, University of Maryland.
- Zhao, H. and Doshi, P. (2007), Haley: A Hierarchical Framework for Logical Composition of Web Services,

BIBLIOGRAPHY

- in: Proceedings of the International Conference on Web Services (ICWS)*, pp. 312–319.
- Zhuo, H., Li, L., Bian, R. and Wan, H. (2007), Requirement Specification Based on Action Model Learning, *in: Proceedings of the International Conference on Intelligent Computing (ICIC)*, volume 4681 of *Lecture Notes in Computer Science*, Springer, pp. 565–574.
- Ziemke, T. (2000), *Situated Neuro-Robotics and Interactive Cognition*, Ph.D. thesis, University of Sheffield.
- Zimmerman, T. and Kambhampati, S. (2003), Learning-Assisted Automated Planning: Looking Back, Taking Stock, Going Forward, *AI Magazine*, volume 24, pp. 73–96.
- Zinkevich, M. and Balch, T. (2001), Symmetry in Markov Decision Processes and its Implications for Single Agent and Multiagent Learning, *in: Proceedings of the International Conference on Machine Learning (ICML)*.
- Zucker, J. D. (2003), A Grounded Theory of Abstraction in Artificial Intelligence, *Phil. Trans. R. Soc. Lond. B.*, volume 358, pp. 1293–1309.

List of Acronyms

ADD	<i>Algebraic Decision Diagram</i>	FODD	<i>First-Order Decision Diagram</i>
ADL	<i>Action Description Language</i>	FOL	<i>First-Order Logic</i>
ADLIN	<i>Adaptive Logic Interpreter</i>	FOLAO*	<i>First-Order LAO*</i>
AI	<i>Artificial Intelligence</i>	FOMDP	<i>First-Order Markov Decision Process</i>
AMBIL	<i>Abstract Model Building using ILP</i>	FOMDP	<i>Fully Observable Markov Decision Process</i>
AMDP	<i>Abstract-state, interval envelope MDP</i>	FORM	<i>First-Order Represented MDP</i>
ANN	<i>Artificial Neural Network</i>	FOVIA	<i>First-Order Value Iteration Algorithm</i>
API	<i>Approximate Policy Iteration</i>	FOX-CS	<i>First-Order X-Classifer System</i>
APRICODD	<i>Approximate Policy Construction using Decision Diagrams</i>	GA	<i>Genetic Algorithm</i>
AR	<i>Adaptive Resolution</i>	GAL	<i>Growing And Learning</i>
ARMS	<i>Action-Relation Modelling System</i>	GCS	<i>Growing Cell Structures</i>
AVI	<i>Approximate Value Iteration</i>	GNG	<i>Growing Neural Gas</i>
BDD	<i>Binary Decision Diagram</i>	GP	<i>Genetic Programming</i>
BLP	<i>Bayesian Logic Program</i>	GPI	<i>Generalized Policy Iteration</i>
BN	<i>Bayesian Network</i>	HASSLE	<i>Hierarchical reinforcement learning based on Subgoal discovery and Subpolicy specialization</i>
CARCASS	<i>Compact Abstraction using Relational Conjunctions for Aggregation of State-action Spaces</i>	HAM	<i>Hierarchies of Abstract Machines</i>
CMAC	<i>Cerebellar Model Articulation Controller</i>	HB	<i>Herbrand Base</i>
CWA	<i>Closed World Assumption</i>	HOL	<i>Higher Order Logic</i>
DBN	<i>Dynamic Bayesian Network</i>	HU	<i>Herbrand Universe</i>
DCS	<i>Dynamic Cell Structures</i>	HMM	<i>Hidden Markov Model</i>
DL	<i>Description Logic</i>	HRL	<i>Hierarchical Reinforcement Learning</i>
DNC	<i>Dynamic Node Creation</i>	HTN	<i>Hierarchical Task Network</i>
DP	<i>Dynamic Programming</i>	IBAL	<i>Integrated Bayesian Agent Language</i>
DR	<i>Deictic Representation</i>	IDP	<i>Intensional Dynamic Programming</i>
DT	<i>Decision Tree</i>	IFP	<i>Inferential Frame Problem</i>
DTP	<i>Decision-Theoretic Planning</i>	IID	<i>Independent and Identically Distributed</i>
DTR	<i>Decision-Theoretic Regression</i>	ILP	<i>Inductive Logic Programming</i>
EA	<i>Evolutionary Algorithm</i>	IPC	<i>International Planning Competition</i>
EBL	<i>Explanation-Based Learning</i>	KB	<i>Knowledge Base(d)</i>
EBRL	<i>Explanation-Based Reinforcement Learning</i>	KBANN	<i>Knowledge-Based Artificial Neural Network</i>
EFUNN	<i>Evolving Fuzzy Neural Networks</i>	KBR	<i>Knowledge-Based Regression</i>
FA	<i>Function Approximation</i>	KR	<i>Knowledge Representation</i>
FFOMDP	<i>Factored First-Order Markov Decision Process</i>	LAMPS	<i>Learning Action Model from Plan Samples</i>
		LBAI	<i>Logic-Based Artificial Intelligence</i>
		LCS	<i>Learning Classifier System</i>
		LGG	<i>Least General Generalization</i>

List of Acronyms

LLLCS	<i>Life-Long Learning Cell Structures</i>	RRL	<i>Relational Reinforcement Learning</i>
LOMDP	<i>LOgical Markov Decision Process</i>	RTD	<i>Relational Temporal Difference</i>
LP	<i>Linear Programming</i>	RTDP	<i>Real-Time Dynamic Programming</i>
LTD	<i>Logical Temporal Difference</i>	RMDP	<i>Relational Markov Decision Process</i>
MACS	<i>Modular Anticipatory Classifier System</i>	ROPG	<i>Relational Online Policy Gradient</i>
MARLIE	<i>Model-Assisted Reinforcement Learning in Expressive Languages</i>	SA	<i>Successive Approximation</i>
MC	<i>Monte Carlo</i>	SARSA	<i>State Action Reward State Action</i>
MDP	<i>Markov Decision Process</i>	SBSA	<i>Set-Based Successive Approximation</i>
MGU	<i>Most General Unifier</i>	SC	<i>Situation Calculus</i>
ML	<i>Machine Learning</i>	SDP	<i>Set-Based Dynamic Programming</i>
MLN	<i>Markov Logic Network</i>	SDP	<i>Symbolic Dynamic Programming</i>
MLP	<i>Multi-Layer Perceptron</i>	SDM	<i>Sparse Distributed Memory</i>
MM	<i>Markov Model</i>	SLAF	<i>Simultaneous Learning and Filtering</i>
MM	<i>Model Minimization</i>	SLD	<i>Linear resolution for Definite clauses with Selection function</i>
MPA	<i>Multi-Part Abstraction</i>	SLDNF	<i>SLD with Negation as Failure</i>
MPI	<i>Modified Policy Iteration</i>	SLP	<i>Stochastic Logic Program</i>
MSE	<i>Mean Squared Error</i>	SMDP	<i>Semi-Markov Decision Process</i>
NAF	<i>Negation As Failure</i>	SPI	<i>Structured Policy Iteration</i>
NN	<i>Nearest Neighbor</i>	SRL	<i>Statistical Relational Learning</i>
NN	<i>Neural Network</i>	SSA	<i>Successor State Axiom</i>
NPPG	<i>Non-Parametric Policy Gradient(s)</i>	SSA	<i>Structured Successive Approximation</i>
OO	<i>Object-Oriented</i>	STRIPS	<i>STanford Research Institute Problem Solver</i>
OWA	<i>Open World Assumption</i>	SVI	<i>Structured Value Iteration</i>
PE	<i>Partial Evaluation</i>	SVRRL	<i>Simultaneous learning of Structure and Value in Relational Reinforcement Learning</i>
PFOL	<i>Probabilistic First-Order Logic</i>	TD	<i>Temporal Difference</i>
PG	<i>Policy Gradient</i>	TILDE	<i>Top-down Induction of Logical Decision Trees</i>
PI	<i>Policy Iteration</i>	TRENDI	<i>TREes aND Instances</i>
PIAGET	<i>Policy Iteration using Abstraction and Generalization Techniques</i>	VFA	<i>Value Function Approximation</i>
PILP	<i>Probabilistic Inductive Logic Programming</i>	VI	<i>Value Iteration</i>
PL	<i>Propositional Logic</i>	VISA	<i>Variable Influence Structure Analysis</i>
PLL	<i>Probabilistic Logic Learning</i>	YACS	<i>Yet Another Classifier System</i>
POMDP	<i>Partially Observable Markov Decision Process</i>		
PRM	<i>Probabilistic Relational Model</i>		
PRS	<i>Predictive Representation of State</i>		
PS	<i>Prioritized Sweeping</i>		
PTS	<i>Probabilistic Transition Structure</i>		
PWLC	<i>Piecewise Linear and Convex</i>		
RBF	<i>Radial Basis Function</i>		
REBEL	<i>Relational Bellman</i>		
REBP	<i>Relational Envelope-Based Planning</i>		
RFP	<i>Referential Frame Problem</i>		
RIB	<i>Relational Instance-Based (Regression)</i>		
RIBL	<i>Relational Instance-Based Learning</i>		
RIBP	<i>Relational Instance-Based Policy</i>		
RL	<i>Reinforcement Learning</i>		
RMPI	<i>Relational Modified Policy Iteration</i>		
RNPC	<i>Relational Nearest Prototype Classification</i>		
RPI	<i>Representation Policy Iteration</i>		

- Abbeel, P. 424
 Abbott, L. F. 37
 Aberdeen, D. 65, 345
 Abul, O. 121, 419, 439
 Agogino, A. 124
 Agre, P. E. 170
 Aha, D. 281
 Ahrns, I. 120, 123, 129
 Al-Ansari, M. A. 97
 Albert, M. 281
 Aler, R. 145
 Alhajj, R. 121, 419, 439
 Allender, E. 101
 Alonso, E. 37
 Alpaydin, E. 120
 Amir, E. 165, 240, 414, 440, 442
 Andersen, C. C. S. 417
 Anderson, C. 204
 Anderson, C. W. 56, 114, 121, 123, 127
 Andre, D. 106, 135, 343, 402
 Apt, K. 186
 Arai, S. 243, 272, 403, 413
 Arkin, R. C. 10, 78, 130, 402
 Arora, S. 101
 Asadi, M. 137, 138, 162, 418
 Asfour, T. 413, 440
 Asgharbeygi, N. 162, 193, 224, 231, 240, 243,
 272, 275, 276, 419
 Ash, T. 119
 Asmuth, J. 23, 165, 218, 220, 225, 441
 Atkeson, C. G. 58, 59, 61, 96, 112, 126, 259,
 260, 263
 Aycenina, M. 219, 417

 Babuška, R. 419, 439
 Bacchus, F. 189, 216, 225
 Back, A. D. 117
 Bäck, T. 123, 285
 Bader, S. 22, 205
 Baillargeon, R. 163
 Bain, M. 60
 Baird, L. C. 127, 129
 Bakker, B. 37, 65, 110, 115, 116, 122, 138
 Bal, H. E. 74, 337
 Balch, T. 98
 Ballard, D. 164, 170
 Baral, C. 207
 Baras, J.S. 96, 325
 Bartlett, P. L. 42

 Barto, A. G. iv, 7, 18, 19, 25, 31, 33, 38, 45, 51,
 52, 54, 56, 61, 62, 66, 70, 84, 87, 95, 98,
 104, 106, 109, 110, 113, 115, 121, 127–129,
 133, 136–139, 270, 342, 343, 390, 409, 467
 Başeski, E. 413, 440
 Bates, E. A. 11, 72
 Batog, A. M. 413, 440
 Baum, E. B. 3, 15, 69, 127, 129, 159, 160, 163,
 169, 227, 232, 239, 298
 Baum, J. 96, 325
 Bazen, A. M. 121, 140, 141
 Bellemare, M. G. 123
 Bellman, R. E. 18, 43, 46, 49, 74, 310
 Bengio, Y. 37
 Benson, S. S. 112, 170, 391, 414, 415
 Bergadano, F. 21, 195, 196, 377
 Bersini, H. 129
 Bertsekas, D. P. 7, 18, 31, 32, 50, 51, 56, 70,
 82, 86, 96, 109, 110, 115, 121, 126, 127,
 241, 313, 325
 Bian, R. 414
 Billard, A. 119
 Bishop, C. M. 21, 106, 108, 115, 116
 Blockeel, H. 20, 126, 178, 190–192, 196, 197,
 203, 204, 219, 223, 224, 233, 236, 238, 242,
 243, 248, 273, 277, 278, 282, 413, 416, 423,
 428, 441, 442
 Blum, A. 147
 Blythe, J. 23, 218, 220
 Bol, R. 186
 Bolle, R. 140
 Bonet, B. 23, 52, 225, 345, 392, 441
 Bongard, M. 160
 Booker, L. B. 123
 Borrajo', D. 440
 Botta, M. 205
 Boutilier, C. 19, 20, 23, 31, 38, 50, 70, 82, 84,
 85, 89, 93, 98, 99, 101–105, 112, 140, 145,
 159, 162, 170, 178, 192, 216, 219–225, 230,
 231, 233–236, 238–242, 317, 324, 325, 329,
 336, 341, 343, 345, 355, 366, 367, 376,
 382–384, 386–389, 402, 434, 435, 438
 Boyan, J. A. 121, 127, 128, 299, 344, 409
 Brachman, R. J. 8, 22, 70, 77, 78, 90, 154, 162,
 163, 177, 188, 206, 211, 214, 217, 218, 331,
 357
 Bradtke, S. J. 52, 390
 Brafman, R. I. 59, 432
 Braitenberg, V. 9

- Bratko, I. 155, 181, 191, 441
 Breiman, L. 125, 202
 Brock, O. 281, 440
 Broda, K. 118
 Broersen, J. 404
 Brooks, R. A. 10, 78, 156, 402
 Browne, A. 205
 Bruske, J. 120, 123, 129
 Bruynooghe, M. 190, 192, 206, 220, 223, 225, 238, 243, 282, 283, 402, 413, 416, 418, 420, 441
 Buçoni, L. 419, 439
 Bulitko, V. 145
 Bundy, A. 336
 Buntine, W. 199, 201, 353
 Burgard, W. 231, 235, 237, 391, 415

 Calbert, G. 83, 145
 Califf, M. E. 190, 202, 299
 Carbonell, J. 104, 336, 342, 346
 Caruana, R. 118, 121
 Cassandra, A. R. 64, 65, 345, 392
 Castañon, D. A. 96, 325
 Castilho, M. 298
 Chang, A. 414
 Chapman, D. 19, 85, 95, 99, 125, 170, 279
 Chellappa, R. 440
 Chen, X. 274, 300
 Chen, X. P. 274
 Chentanez, N. 66
 Choi, D. 403
 Chongkasemwongse, K. 205
 Choueiry, B. Y. 79
 Cios, K. 119, 120
 Claplin, H. 8, 156
 Clark, A. 78, 111, 155, 156, 170
 Cloete, I. 118
 Clore, G. L. 406
 Cocora, A. 231, 235, 237, 391, 415
 Cohen, A. 169, 387
 Cohen, P. R. 112, 414
 Cohn, A. G. 440
 Cohn, D. 85
 Cole, J. 188, 192, 234, 278, 430
 Collins, A. 406
 Colombetti, M. 60, 129
 Copeland, G. 380
 Coradeschi, S. 440
 Crites, R. H. 109, 121, 409

 Croonenborghs, T. 162, 190, 192, 206, 220, 223–225, 238, 240, 243, 279, 282, 283, 402, 413, 416, 418–420, 441
 Cumby, C. 204
 Czarnecki, C. A. 63, 122

 Dabney, W. 167, 168, 171, 172, 222, 233, 236, 238, 280, 428, 440, 442
 Dai, P. 228
 Damasio, A. R. 406
 Dastani, M. 161, 219, 234, 242, 243, 398, 402–406
 d'Avila Garcez, A. S. 118
 Davis, L. S. 440
 Davis, M. 11, 13, 15, 154
 Dayan, P. 19, 37, 55, 56, 99, 132, 408
 de Boer, F. S. 404, 410
 de Jong, E. D. 78
 de la Rosa, T. 390
 De Raedt, L. 20–22, 81, 89, 126, 157, 161, 166, 167, 169, 191–197, 202–205, 220, 221, 224–227, 230, 231, 233–237, 239, 240, 242, 243, 248, 252, 264, 272, 273, 277, 278, 285, 337, 348, 371, 377, 382, 385, 391, 402, 415, 423, 428, 437–440, 442
 de Salvo Braz, R. 240
 de Schutter, B. 419, 439
 de Wolf, R. 195, 197, 198, 348
 Dean, T. 19, 23, 31, 38, 48, 50–52, 70, 82, 84, 85, 93, 95–102, 104, 133, 136, 273, 317, 321, 324, 325, 327, 328, 333–335, 344, 376, 384
 Dearden, R. W. 19, 84, 89, 93, 96, 98, 101–106, 112, 140, 147, 222, 329, 334, 336, 341, 343–345, 376, 385
 Degris, T. 19, 106
 Dehaspe, L. 178, 196, 203
 Demoen, B. 178, 196, 197, 203
 Dempster, A. P. 109, 139
 Denecker, M. 194
 Dennett, D. C. 5, 14, 153, 158, 209, 211
 Detry, R. 413, 440
 Devin, V. E. 440
 Dietrich, E. 10, 78
 Dietterich, T. G. 18, 19, 83, 98, 104–106, 125, 126, 128, 129, 134–136, 140, 147, 158, 190, 204, 234, 273, 327, 334, 337, 341–343, 401, 402
 Digney, B. 138
 Dignum, F. 403–405

- Dijkstra, E. W. 335
Dillmann, R. 413, 440
d'Inverno, M. 404
Diuk, C. 19, 82, 83, 106, 133, 136, 145, 169, 387
Divina, F. 22, 155, 203, 287, 434
Dixon, K. R. 60
Doan, A. 95
Domingos, P. 204, 205, 435
Dorigo, M. 60, 129
Doshi, P. 391
Doyle, P. 165, 440
Drescher, G. 11, 37, 61, 66, 72, 78, 112, 147
Driessens, K. 162, 167, 192, 193, 206, 219, 222, 224, 225, 227, 230, 231, 233, 234, 236–241, 244, 248, 251, 252, 270, 277–279, 281–283, 301, 371, 377, 385, 413, 415, 416, 418, 419, 423
Džeroski, S. 20–22, 126, 157, 162, 166, 178, 180, 195, 196, 199, 203, 224, 225, 227, 228, 230, 231, 233, 234, 236–239, 242, 243, 247, 248, 252, 270, 273, 277, 278, 281, 371, 377, 385, 402, 415, 423, 428, 442, 448, 451, 458, 459, 475
Edelkamp, S. 218, 225
El Fallah-Seghrouchni, A. 420
Ellman, T. 325, 336, 342, 346
Elman, J. L. 11, 72, 116
Engel, Y. 126
Epelman, M. A. 96, 325
Erdmann, M. 324
Ernst, D. 111, 126
Erol, K. 22, 130
Etzioni, O. 104, 336, 342, 346
Even-Dar, E. 96
Fagin, R. 155, 189, 412, 419, 441
Fahlman, S. E. 117, 119, 120, 123
Feng, C. 200
Feng, Z. 82, 96, 104, 140, 334, 344, 345, 390
Ferber, J. 23, 77, 155, 243, 402, 419
Ferguson, D. 52
Fermüller, C. G. 198, 221, 233, 349
Fern, A. 23, 128, 165, 188, 224, 225, 227, 230, 233–238, 240, 242, 243, 250, 299, 300, 302, 329, 386, 390, 391, 415, 416, 438
Fernández, F. 281, 390, 414, 419
Ferns, N. 96
Ferrein, A. 165, 216, 440
Fiesler, E. 119, 120
Fikes, R. E. 22, 76, 101, 112, 207, 212, 308
Finney, S. 15, 126, 154, 159, 160, 162, 164, 170, 171, 219, 242, 247, 280, 442
Finton, D. J. 77, 84, 85, 93, 95, 99, 145, 147
Finzi, A. 22, 216, 228, 329, 342, 402, 420, 439
Fischer, J. 375, 376, 382, 385
Fitch, R. 83, 145
Flach, P. A. 170, 195, 196, 204
Flann, N. S. 98, 104–106, 327, 334, 337, 341–343
Floreano, D. 10, 129, 285
Fodor, J. A. 156
Fox, D. 440
Främling, K. 57, 60
Frege, G. 15, 154, 172
French, R. M. 108, 117, 121
Friedman, J. H. 21, 106, 107, 125, 155, 202, 205
Friedman, N. 89, 106, 109, 204, 236, 343, 387
Fritz, C. 216
Fritzke, B. 119, 120, 123
Frühwirth, T. W. 353, 366
Gabaldon, A. 216
Gabbay, D. M. 118
Galavotti, M. C. 21, 189
Gamut, L. T. F. 189
Gao, Y. 274, 300
Garcia, A. 440
García-Durán, R. 281, 390, 419
Gärdenfors, P. 55, 78, 156, 163, 165, 207
Gardiol, N. H. 15, 126, 154, 159, 160, 162, 164, 170, 171, 219, 220, 222, 223, 225, 226, 234, 238, 242, 247, 280, 283, 390, 391, 413, 415, 423, 442
Garlick, R. 124
Gärtner, T. 22, 167, 193, 204, 222, 234, 238, 282, 413
Ge, S. 274, 300
Gearhart, C. 165, 167, 172, 190, 206, 219, 223–225, 228, 235, 238, 239, 241, 243, 250, 297, 298, 387, 416, 437, 438, 440
Geffner, H. 23, 52, 82, 191, 214, 224, 225, 229, 237, 264, 299, 345, 390–392, 441
Geib, C. 413, 440
Geibel, P. 413
Gelfond, M. 22, 206, 207, 214
Georgeff, M. P. 404
Gerard, P. 282

Author Index

- Gerez, S. H. 121, 140, 141
Getoor, L. 22, 203, 204, 387, 452
Geurts, P. 111, 126
Ghahramani, Z. 85, 100
Ghallab, M. 22, 214, 218
Ghassem-Sani, Gh. 415
Ghavamzadeh, M. 129, 136
Gil, Y. 104, 336, 342, 346, 414, 415
Giles, C. L. 118, 121
Giordana, A. 205
Giunchiglia, F. 79
Givan, R. 19, 23, 84, 93, 95–98, 104, 114, 128, 136, 148, 162, 165, 188, 219, 224, 225, 227, 230, 233–238, 240, 242–244, 248, 250, 273, 299, 300, 302, 321, 327, 329, 333–335, 344, 376, 386, 389–391, 415, 416, 438
Glanz, F. H. 114
Glaubius, R. 96
Goebel, R. 3, 21, 177, 178
Goetschalckx, R. 243, 279, 402, 420
Goldberg, D. E. 123, 203, 285
Goldbloom Bloch, W. iii
Goldsmith, J. 82, 228
Goldszmidt, M. 19, 89, 98, 102–104, 112, 140, 329, 336, 341, 343, 345, 376
Gordon, G. J. 52, 65, 94, 110, 126, 273, 438, 442
Görtz, G. 3, 70, 154
Gottlob, G. 198, 221, 233, 349
Grant, T. J. 413
Grefenstette, J. J. 19, 65, 86, 124, 129, 286, 298
Greig, M. 84, 93, 96–98, 104, 136, 273, 321, 327, 333, 335, 344, 376
Greiner, R. 145, 250
Gretton, C. 224, 230, 236, 239, 242, 300, 379, 386, 389, 390, 392, 431, 442
Groen, F. 419, 439
Groote, J. F. 192, 382
Grosskreutz, H. 216
Großmann, A. 58, 65, 99, 112, 123, 147, 225, 382
Grounds, M. 381, 417
Gu, D. 419, 439
Gu, Y. 402
Guestrin, C. 105, 165, 167, 172, 190, 206, 219, 223–225, 228, 235, 238, 239, 241, 243, 250, 297, 298, 387, 388, 416, 437, 438, 440
Gunetti, D. 21, 195, 196, 377
Guo, H. F. 376
Gupta, G. 376
Gupta, N. 164
Haddawy, P. 95
Haddon, M. 5
Halbritter, F. 413
Halpern, J. Y. 21, 155, 189, 204, 216, 225, 412, 419, 437, 441
Hamker, F. H. 120
Handley, S. 124
Hanks, S. 19, 23, 31, 34, 38, 50, 70, 82, 93, 99, 101, 102, 222, 317, 324, 325, 350, 384
Hansen, E. A. 52, 104, 140, 390
Hanson, S. J. 119
Harada, D. 60
Harnad, S. 9, 78, 156, 440
Hastie, T. 21, 106, 107, 155, 205
Haugeland, J. 3
Hayes, G. 119
Hayes, P. J. 147, 155
Haykin, S. 21, 108, 113–117
Heinke, D. 120
Helmert, M. 164
Hendler, J. 22, 130
Hengst, B. 19, 83, 95, 139, 145
Hernandez, A. G. 420
Hertzberg, J. 441
Hindriks, K. V. 404, 410
Hinton, G. E. 19, 85, 99, 106, 132
Hitzler, P. 22, 205
Hoey, J. 102, 104, 105, 341, 382, 389
Hoffman, J. 218, 225
Hofstadter, D. R. 3, 160
Hogg, D. C. 440
Holland, J. H. 4, 6, 18, 123, 285
Hölldobler, S. 22, 23, 205, 225, 355, 358, 382, 384, 390
Holmes, M. P. 66
Holte, R. C. 79
Hommel, B. 413, 440
Honavar, V. 119, 123
Hong, L. 140
Horváth, T. 192
Howard, R. A. 18, 46, 48
Howe, A. E. 22, 125, 214, 218
Hoze, A. 138
Hu, A. 102, 104, 341, 382, 389
Huber, M. 137, 138, 162, 418
Hulstijn, J. 404

- Humphreys, D. 298
- Ibarüengoytia, P. 273
- Irodova, M. 169
- Isasi, P. 145
- Isbell jr., C. L. 66
- Itoh, H. 242, 279, 300, 442
- Jaakkola, T. 19, 93, 94, 99, 126, 230, 259, 327, 428
- Jacobs, N. 196, 197, 203
- Jacobs, S. 165, 440
- Jain, A. K. 140, 141
- James, M. R. 65
- Jang, J. S. R. 21, 118
- Janssens, G. 178
- Jennings, N. R. 4, 77, 398
- Jensen, D. 204
- Jiang, X. 140
- Jiang, Y. 414
- Jiménez, S. 414
- Johnson, M. H. 11, 72
- Jones, D. T. 160, 178
- Jong, N. K. 139
- Jonsson, A. 19, 106, 133, 139
- Jordan, M. I. 19, 93, 94, 99, 109, 112, 126, 167, 230, 259, 327, 428
- Joshi, S. 192, 223, 233, 355, 382–384, 389
- Jozefowicz, J. 37
- Kadie, C. M. 413
- Kaelbling, L. P. 7, 15, 16, 18, 19, 31, 48, 50–52, 59, 64, 65, 72, 82, 83, 85, 95, 96, 99, 112, 125, 126, 133, 145, 154, 159, 160, 162–164, 170, 171, 219, 220, 222, 225, 226, 238, 241–243, 247, 279, 280, 328, 345, 390–392, 414, 415, 442
- Kahneman, D. 1
- Kaikhah, K. 124
- Kakade, S. M. 75, 82
- Kalaska, J. 37
- Kalkan, S. 413, 440
- Kambhampati, S. 23, 218, 243, 415
- Kanawaza, K. 85, 100
- Kanodia, N. 165, 167, 172, 190, 206, 219, 223–225, 228, 235, 238, 239, 241, 243, 250, 297, 298, 387, 416, 437, 438, 440
- Karabaev, E. 23, 218, 223, 234, 239, 329, 355, 358, 374, 382, 385, 390
- Karalic, A. 191
- Karmiloff-Smith, A. 11, 72
- Kasabov, N. K. 118, 120
- Katz, D. 281, 440
- Kaufman, M. 403
- Kautz, H. 440
- Kearns, M. 59, 101, 106
- Keerthi, S. S. 18, 109
- Keller, P. W. 19, 148
- Kersting, K. 22, 89, 157, 162, 167, 191, 197, 203–205, 219–221, 224–226, 230, 231, 233–235, 237, 240, 243, 244, 252, 264, 272, 285, 301, 381, 382, 391, 402, 415, 438, 439
- Khardon, R. 128, 191, 192, 223, 224, 233, 235, 237, 238, 242, 299, 355, 382–386, 389–391
- Khoshafian, S. 380
- Khosla, P. K. 60
- Kibler, D. 281
- Kietz, J. U. 198, 233, 348
- Kijsirikul, N. 205
- Kim, K. E. 95, 98, 334
- Kimmig, A. 204, 337
- Kinny, D. 404
- Kirman, J. 51, 52
- Kirsten, M. 192
- Kjær-Nielsen, A. 413, 440
- Klein, U. 138
- Knoblock, C. A. 22, 104, 214, 218, 336, 342, 346
- Kochenderfer, M. J. 298
- Koenig, S. 42, 145, 223
- Kok, J. R. 85, 412, 419
- Kok, S. 205
- Kokar, M. M. 31
- Koller, D. 105, 106, 165, 167, 172, 190, 204, 206, 219, 223–225, 228, 235, 238, 239, 241, 243, 250, 297, 298, 387, 388, 416, 424, 437, 438, 440
- Konda, V. 56
- Konidaris, G. 60
- Koop, A. 66
- Korf, R. E. 11, 79–81, 88, 98, 147
- Kraft, D. 413, 440
- Kraft, L. G. 114
- Kragic, D. 440
- Kramer, S. 170, 191, 204
- Kress, M. 298
- Kretchmar, R. 114
- Kripke, S. A. 189
- Kröse, B. 419, 439

Author Index

- Krueger, V. 440
Krüger, N. 413, 440
Kubat, M. 250
Kudenko, D. 381, 417
Kunzle, L. A. 298
Kuokka, D. R. 104, 336, 342, 346
Kuppili, P. 419
Kushmerick, N. 23, 34, 222
Kwok, T. Y. 119, 120
- Lachiche, N. 204
Lagoudakis, M. G. 86, 112, 128, 391
Laird, J. E. 336, 398, 403, 439
Laird, N. M. 109, 139
Lakemeyer, G. 165, 216, 440
Lambert III, T. J. 96, 325
Landwehr, N. 440
Lane, T. 96, 228, 416
Lang, J. 211
Langford, J. 128
Langley, P. 9, 21, 70, 124, 147, 193, 224, 231, 243, 272, 275, 276, 398, 399, 403, 413, 419, 439
Lanzi, P. L. 65, 112, 129, 280, 297
Lavrac, N. 21, 22, 166, 170, 178, 180, 195, 196, 199, 203, 228, 448, 451, 458, 459, 475
Lawrence, S. 117, 118, 121
Leach, S. 98
Lebiere, C. L. 117, 119, 120, 123
Lecheta, E. 298
Lecoeuche, R. 191, 230, 231, 235, 237, 278, 298, 299, 302, 390
Lehtokangas, M. 120
Lesperance, Y. 22, 216, 402, 419
Letia, I.A. 419
Levesque, H. J. 8, 22, 70, 77, 78, 90, 154, 162, 163, 177, 188, 206, 211, 214, 216–218, 225, 331, 357, 402, 419
Levine, J. 298
Levner, I. 145
Li, L. 93, 94, 114, 126, 148, 230, 239–241, 345, 414, 418
Liao, L. 440
Lifschitz, V. 22, 155, 172, 206, 207, 213, 214
Likhachev, M. 52
Lin, F. 211, 217
Lin, K. J. 19, 65, 112, 115, 121
Lin, S. H. 133
Littman, M. L. 7, 18, 19, 23, 31, 48, 50, 64, 65, 72, 78, 82, 83, 93, 94, 101, 106, 114, 126, 133, 136, 145, 148, 165, 169, 218–220, 222, 225, 230, 239–241, 328, 344, 345, 387, 391, 392, 409, 414, 418, 438, 441
- Liu, Y. 42, 145, 223
Lloyd, J. W. 21, 181, 188, 192, 202, 234, 239, 278, 379, 389, 430
Lorenzo, D. 414
Lozano-Perez, T. 324
Lübbe, M. 198, 233, 348
Luck, M. 404
Luger, G. F. 3, 70, 154
Lukasiewicz, T. 22, 216, 228, 329, 342, 402, 420, 439
- Mackworth, A. 3, 21, 177, 178
Maclin, R. 170, 192, 193, 222, 231, 235, 238, 274, 275, 280, 282, 302, 418, 419
Madani, O. 145
Magee, D. R. 440
Maggioni, M. 20, 148, 435
Mahadevan, S. 19, 20, 42, 57, 126, 129, 136, 139, 146, 148, 435
Maio, D. 140
Majercik, S. M. 82
Makino, T. 66
Malak, M. J. 60
Maloberti, J. 348
Maloof, M. A. 36, 109, 233, 250
Maltoni, D. 140
Maluszinski, J. 155, 184–186
Manfredi, V. 139
Manna, Z. 335
Mannor, S. 19, 126, 138, 148
Mansour, Y. 19, 48, 50, 96, 128, 144, 328
Marecki, J. 345
Margolis 8, 78, 156, 163, 165, 444
Markman, A. B. 8, 10, 70, 77, 78, 90, 154, 156, 162
Markovitch, S. 147
Marks II, R. J. 21, 111, 115–118, 120, 123
Marquis, P. 211
Martens, B. 194
Martin, M. 191, 224, 229, 237, 264, 299, 390, 391
Mason, M. M. 324
Mataric, M. J. 60
Matthews, W. H. 40
Mausam 219, 225, 231, 235, 237, 239, 243, 391
McAllester, D. A. 19, 128, 144, 211, 217, 230

- McCallum, A. K. 439
 McCallum, R. A. 99, 125, 164, 280
 McCarthy, J. 22, 147, 155, 172, 214, 382, 402, 419
 McDermott, D. V. 22, 101, 214, 218, 350
 McGovern, A. 138, 167, 168, 171, 172, 222, 233, 236, 238, 280, 428, 440, 442
 McMahan, H. B. 52
 Mehta, N. 140
 Meir, R. 126
 Mellor, D. 22, 162, 229, 236, 240, 255, 280, 284, 297, 423
 Menache, I. 138
 Metta, G. 398
 Meuleau, N. 96, 104, 129, 334, 344, 345
 Meyer, J. A. 112
 Meyer, J. J. 161, 219, 234, 242, 243, 398, 402–404, 406, 410
 Miikkulainen, R. 124, 129, 286, 296
 Miller, W. T. 114
 Minker, J. 3, 155, 453
 Minsky, M. 3, 6, 11, 113, 162
 Minton, S. 104, 336, 342, 346
 Mitchell, M. 123, 285
 Mitchell, T. M. 6, 21, 70, 125, 155, 194, 336
 Miyamoto, Y. 113
 Mizutani, E. 21, 118
 Mondragón, E. 37
 Mooney, R. J. 190, 202, 299
 Moore, A. W. 7, 18, 31, 58, 59, 61, 72, 96, 97, 112, 121, 126–129, 259, 260, 263, 299
 Moore, C. 101
 Morales, E. F. 221, 233, 235, 240, 252, 272–274, 391, 413, 415, 428
 Moriarty, D. E. 19, 65, 86, 124, 129, 286, 296, 298
 Moses, Y. 155, 189, 412, 419, 441
 Mourão, K. 413, 440
 Mueller, E. T. 22, 154, 156, 158, 206, 207, 209–211, 214, 217, 218
 Muggleton, S. H. 21, 81, 195, 196, 200, 202
 Muller, T. J. 229, 230, 236, 285, 292, 297
 Munos, R. 96, 97
 Murphy, K. 65, 204
 Murtagh, S. 337

 Nakamura, K. 242, 279, 300, 442
 Namihira, M. 96
 Nason, S. 336, 403, 439
 Natarajan, S. 419

 Nau, D. S. 22, 130, 164
 Needham, C. J. 440
 Nejati, N. 243, 272, 403
 Ng, A. Y. 60
 Ng, K. S. 188, 192, 234, 278, 430
 Nicholson, A. 51, 52, 96, 325
 Nienhuys-Cheng, S. H. 195, 197, 198, 348
 Nilsson, N. J. 3, 4, 22, 76, 101, 112, 189, 207, 212, 308, 335, 358, 398
 Nilsson, U. 155, 184–186
 Niranjana, M. 19, 56, 115, 121
 Nolfi, S. 10, 129, 285
 Norvig, P. 3, 4, 18, 22, 52, 65, 70, 102, 130, 154, 162, 163, 188, 206, 211, 212, 216–218, 335, 336, 357

 Oates, T. 15, 112, 126, 154, 159, 160, 162, 164, 170, 171, 219, 242, 247, 280, 414, 442
 Ollington, R. 123
 Olshen, R. A. 125, 202
 Omar, R. 156, 178
 Orengo, C. A. 160, 178
 Ormoneit, D. 126
 Ortiz, J. 440
 Ortony, A. 406
 Otero, R. P. 414

 Painter-Wakefield, C. 114, 148
 Palodeto, V. 298
 Panangaden, P. 96
 Pankanti, S. 140
 Papavassiliou, V. A. 126
 Papert, S. A. 113
 Papudesi, V. 162, 418
 Parekh, R. 119, 123
 Parisi, D. 11, 72
 Parr, R. 19, 86, 105, 106, 112, 114, 128, 135, 148, 343, 388, 391
 Parsons, S. 411
 Pasula, H. M. 16, 112, 163, 222, 238, 241, 243, 414, 415
 Patrascu, R. 105, 114, 387
 Pearl, J. 4
 Pednault, E. 22, 207, 214, 335
 Peng, J. 61
 Peshkin, L. 129
 Peterson, T. 118, 122, 205
 Petrick, R. 413, 440
 Petrick, R. P. A. 413, 440
 Pfeffer, A. 225, 402

Author Index

- Pfeifer, R. 3, 9, 10, 66, 78, 156, 210
Piaget, J. 11, 72, 88, 119, 147, 163
Piater, J. 413, 440
Picard, R. W. 406
Pickett, M. 138
Pineau, J. 65, 442
Piola, R. 205
Plaat, A. 3, 34
Plagemann, C. 231, 235, 237, 391, 415
Plotkin, G. D. 21, 197, 198
Plunkett, K. 11, 72
Poel, M. 121, 140
Polat, F. 121, 419, 439
Poli, R. 123
Poole, D. 3, 21, 23, 177, 178, 225, 336, 345, 402
Popović, M. 413, 440
Porto, V. W. 119, 123
Potts, D. 139
Powell, W. B. 18, 467
Prechelt, L. 119, 120
Precup, D. 19, 72, 96, 105, 113, 115, 119, 123, 130, 133, 137, 148, 170, 419
Price, B. 20, 23, 85, 221, 223, 225, 233, 234, 329, 355, 366, 367, 382, 383, 402
Pugeault, N. 413, 440
Puterman, M. L. 18, 31, 38, 51, 131
Pyeatt, L. D. 125
Pylyshyn, Z. W. 156
Pyuro, Y. 281, 440

Quartz, S. R. 11, 72, 119
Quinlan, J. R. 196, 202
Quinlan, P. T. 119

Rabin, S. 440
Rafols, E. J. 66
Raiko, T. 204
Ram, A. 22, 96, 114, 214, 218
Rammé, G. 223, 385
Ramon, J. 22, 155, 162, 167, 178, 190–193, 196, 203, 220, 222–225, 230, 231, 233, 234, 236–238, 240, 241, 243, 251, 277–279, 281–283, 371, 402, 413, 419, 420, 438, 441
Rao, A. S. 404
Ratitch, B. 54, 59, 60, 110, 115, 126
Ravindran, B. 18, 84, 95, 98, 109, 136, 137
Ray, S. 140
Reed, R. D. 21, 111, 115–118, 120, 123
Reid, M. 417

Reiter, R. 20, 22, 23, 85, 154, 206, 207, 210, 211, 213–221, 223, 225, 230, 233, 234, 329, 335, 337, 355, 366, 367, 382, 383, 399, 402, 419, 441
Rennie, J. 439
Reveliotis, S. A. 31
Reyes, A. 273
Reynolds, S. I. 54, 61, 99
Richards, N. 124
Richardson, M. 204, 205
Ridens, M. 416
Riedmiller, M. 122
Ring, M. B. 122
Rivest, F. 37, 120, 123
Rivest, R. L. 350
Rodrigues, C. 282
Rogers, S. 398
Rollinger, C. R. 3, 70, 154
Romein, J. W. 74, 337
Roncagliolo, S. 219, 220, 225, 417
Roth, D. 204, 240
Rouveirol, C. 282
Roy, N. 96
Rubin, D. B. 109, 139
Rudary, M. R. 65
Rummery, G. A. 19, 56, 115, 121
Russell, A. 101
Russell, S. J. 2–4, 18, 19, 22, 52, 60, 65, 70, 80, 82, 102, 126, 130, 135, 154, 162, 163, 188, 206, 211, 212, 216–218, 335, 336, 357, 402
Ryan, M. R. K. 19, 129, 136, 417
Rylatt, R. M. 63, 122

Saad, E. 272
Sablon, G. 413
Safaei, J. 415
Saffiotti, A. 440, 441
Saitta, L. 79, 80, 147
Sallans, B. 85, 106
Sammur, C. 60
Samuel, A. L. 18, 38, 109, 343
Sandini, G. 398
Sanghai, S. 204
Sanner, S. 170, 178, 192, 219, 224, 225, 231, 233, 235, 236, 238–241, 280, 282, 381, 386–389, 423, 433–435, 438
Santamaria, J. C. 96, 114
Santos, M. V. 358
Santos, P. E. 440
Sato, T. 243, 272

- Saxena, S. 104, 342, 343
 Scarselli, F. 117, 124
 Schaal, S. 126
 Schaeffer, J. 3, 34
 Scheier, C. 3, 9, 10, 66, 78, 156, 210
 Schipper, H. 141
 Schmid, U. 163
 Schmidhuber, J. H. 19, 59, 61, 65, 136, 138, 286
 Schmolze, J. 214, 345, 392, 441
 Schneeberger, J. 3, 70, 154
 Schölkopf, B. 21, 155, 193
 Scholz, J. 83, 145
 Schultz, A. C. 19, 65, 86, 124, 129, 286, 298
 Schuurmans, D. 105, 114, 387
 Schwartz, A. 19, 57, 110, 121, 127, 130, 138
 Schwind, C. 206, 209, 211
 Scott, P. D. 147
 Sebag, M. 348
 Seda, A. K. 205
 Seese, D. 298
 Sejnowski, T. J. 11, 72, 119
 Sen, S. 126
 Seppi, K. D. 50
 Ser, W. 140
 Shahaf, D. 414, 442
 Shapiro, D. 403, 413
 Shapiro, E. 155, 181
 Shavlik, J. 118, 170, 192, 193, 222, 231, 235, 238, 274, 275, 280, 282, 302, 418, 419
 Shen, W. M. 413
 Shimkin, N. 138
 Shin, M. C. 51
 Shultz, T. R. 72, 119, 120
 Si, J. 18, 467
 Sigaud, O. 19, 106, 112
 Silva, F. 298
 Simari, G. I. 411
 Simmons, R. 65, 442
 Simsek, O. 138
 Singh, S. 19, 48, 50, 52, 59, 65, 66, 78, 85, 93, 94, 99, 105, 106, 126, 128, 130, 133, 137, 144, 230, 259, 327, 328, 390, 391, 419, 428, 442
 Sinthupinyo, S. 205
 Skvortsova, O. 23, 198, 218, 223, 225, 233, 234, 239, 329, 348, 355, 358, 374, 382, 384, 385, 390
 Slaney, J. 163, 164, 263, 439
 Sloan, R. H. 82, 169
 Smart, W. D. 94, 96, 110, 126
 Smith, R. L. 96, 325
 Smith, T. 65, 442
 Smola, A. J. 21, 155, 193
 Smolensky, P. 118
 Soldano, H. 420
 Sommer, G. 120, 123, 129
 Song, Z. W. 274
 Soni, V. 391, 442
 Soutchanski, M. 216, 220, 230, 402
 Sowa, J. F. 8, 70, 77, 90, 154, 156, 162, 165
 Spaan, M. T. J. 65, 345, 412, 441, 442
 Spott, M. 122
 Spruit, P. A. 211
 Srinivasan, A. 170, 178, 239, 275
 St-Aubin, R. 102, 104, 105, 341, 382, 389
 Stanley, K. 124
 Steedman, M. 413, 440
 Steinkraus, K. 83, 145
 Stentz, A. 52
 Sterling, L. 155, 181
 Stevens, S. 416
 Stolzmann, W. 112, 129
 Stone, C. J. 125, 202
 Stone, P. 129, 139, 418, 419, 439
 Stracuzzi, D. J. 72, 81, 113, 119, 162, 170, 193, 224, 231, 240, 275, 276, 419
 Strehl, A. 19, 82, 83, 106, 133, 136, 145
 Struyf, J. 202, 203, 233
 Subrahmanian, V. S. 440
 Suc, D. 83, 145
 Sucar, L. E. 273
 Sun, C. T. 21, 118
 Sun, R. 118, 119, 122, 205
 Sutton, R. S. iv, 7, 18, 19, 25, 31, 33, 38, 45, 51, 54–56, 58, 61, 62, 65, 66, 70–72, 78, 87, 96, 105, 110–115, 121, 127, 128, 130, 133, 137, 144, 270, 275, 343, 391, 419, 442
 Szepesvari, C. 94, 110
 Tadepalli, P. 105, 136, 140, 162, 219, 220, 225, 244, 248, 342, 417
 Takagi, T. 66
 Tambe, M. 345
 Tan, M. 410
 Tanner, B. 66
 Tash, J. 52
 Taskar, B. 22, 203, 204, 387, 424, 452
 Taylor, M. E. 129

- Taylor, R. H. 324
 Teichteil-Koenigsbuch, F. 218
 Tennenholtz, M. 59
 Tesauro, G. J. 109, 121
 Theocharous, G. 136
 Thiébaux, S. 163, 164, 224, 230, 236, 239, 263,
 300, 379, 386, 389, 390, 431, 439
 Thielscher, M. 22, 207, 216, 217, 382, 402
 Thon, I. 440
 Thorndike, E. L. 37
 Thornton, C. 11, 72, 78, 91, 111, 155, 170, 424
 Thornton, J. M. 160, 178
 Thornton, S. 11, 72
 Thrun, S. 19, 65, 96, 110, 121, 127, 130, 138,
 216, 220, 230, 402, 442
 Tibshirani, R. 21, 106, 107, 155, 205
 Toivonen, H. 204, 337
 Topol, Z. 345
 Torrey, L. 274, 418, 419
 Towell, G. G. 118
 Tran, S. C. 207
 Tran, S. D. 440
 Tsitsiklis, J. 7, 18, 31, 50, 51, 56, 70, 82, 86,
 109, 110, 115, 121, 126, 127, 241, 313
 Tsoi, A. C. 116, 117, 124
 Turaga, P. 440
 Tuyls, K. 220, 225, 243, 402, 420
 Tveretina, O. 192, 382
 Tversky, A. 1
- Ude, A. 440
 Udrea, O. 440
 Uehara, K. 113
 Utgoff, P. E. 72, 81, 104, 113, 119, 125, 126,
 170, 279, 337, 342, 343
 Uther, W. T. B. 125
- Vamplew, P. 123
 van der Hoek, W. 404, 410
 van der Meulen, P. G. M. 141
 van der Torre, L. 404
 van der Voort van der Kleij, G. 110, 122
 van Harmelen, F. 336
 van Laer, W. 90, 160–162, 166, 193, 196, 197,
 202–204, 242, 348, 428
 van Otterlo, M. v. 121, 140, 159, 161, 162, 191,
 219–221, 224, 225, 229–231, 233, 234, 236,
 240, 242–244, 247, 248, 252, 272, 283, 285,
 297, 382, 398, 402, 413
 van Riemsdijk, B. 405
- van Roy, B. 127
 Vandecasteele, H. 178
 Vardi, M. Y. 155, 189, 412, 419, 441
 Veloso, M. 22, 125, 214, 218, 419, 439
 Venkataraman, S. 105, 388
 Vere, S. A. 413
 Vernon, D. 398
 Vollbrecht, H. 99
- Waldinger, R. 216, 314, 335
 Walker, T. 170, 192, 193, 222, 231, 235, 238,
 274, 275, 280, 282, 302, 418, 419
 Walsh, T. 79
 Walsh, T. J. 93, 94, 126, 230, 239–241, 414,
 418, 438
 Wan, H. 414
 Wang, C. 192, 214, 223, 233, 345, 347, 355,
 382, 383, 385, 386, 392, 423, 441, 442
 Wang, W. 274, 300
 Wang, X. 125, 126, 129, 414, 415
 Wang, Y. 403
 Washington, R. 96, 104, 334, 344, 345
 Watkins, C. J. C. H. 19, 55, 56, 61, 408
 Wehenkel, L. 111, 126
 Weisbrod, J. 122
 Weiss, G. 23, 77, 155, 243, 402, 410, 419
 Weissman, D. 23, 165, 218, 220, 225, 441
 Weld, D. S. 22, 23, 34, 204, 207, 214, 218, 219,
 222, 225, 231, 235, 237, 239, 243, 391
 Wermter, S. 119
 Westermann, G. 11, 119, 120
 Whitehead, S. D. 111, 121, 164, 170, 275
 Whiteson, S. 129
 Widmer, G. 191, 250
 Wiering, M. A. 19, 54, 59–62, 65, 94, 110, 112,
 126, 136, 138, 161, 219, 234, 242, 243, 260,
 261, 263, 267, 398, 402, 409, 419, 439
 Wilkins, D. 22, 214, 218
 Williams, R. J. 19, 61, 97, 123, 128
 Wilson, A. 96, 228, 416, 438
 Wilson, S. W. 112, 129
 Wingate, D. 50, 391, 442
 Winston, W. L. 32
 Witkowski, C. M. 37
 Witten, I. H. 56
 Wolfe, A. P. 138
 Wolfe, B. 391, 442
 Wolpert, D. H. 81
 Wooldridge, M. 4, 23, 77, 155, 158, 189, 219,
 243, 398, 402, 404, 420, 441

- Wörgötter, F. 413, 440
Wrobel, S. 180, 192
Wu, J. H. 114, 148, 389, 390
Wu, K. 414
Wuillemin, P. H. 19, 106
Wunsch, D. 18, 467
Wynne-Jones, M. 119
- Yang, E. 419, 439
Yang, J. 119, 123
Yang, Q. 414
Yao, X. 123
Yau, W. Y. 140
Yee, R. C. 104, 342, 343
Yeung, D. Y. 119, 120
Yoon, S. W. 23, 128, 165, 188, 224, 225, 227,
230, 233–238, 240, 242, 243, 250, 299, 300,
302, 329, 386, 390, 391, 415, 416, 423, 438
Younes, H. L. S. 23, 165, 218–220, 222, 225,
441
- Zadrozny, B. 128
Zettlemoyer, L. S. 16, 112, 163, 222, 238, 241,
243, 414, 415
Zhang, C. 96, 325
Zhao, H. 391
Zhuo, H. 414
Ziemke, T. 78, 122
Zilberstein, S. 52, 104, 140
Zimmerman, T. 23, 218, 243, 415
Zinkevich, M. 98
Zucker, J. D. 79, 81
Zurada, J. M. 118