A CAUSAL APPROACH TO HIERARCHICAL DECOMPOSITION IN REINFORCEMENT LEARNING

A Dissertation Presented

by

ANDERS JONSSON

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2006

Department of Computer Science

© Copyright by Anders Jonsson 2006

All Rights Reserved

A CAUSAL APPROACH TO HIERARCHICAL DECOMPOSITION IN REINFORCEMENT LEARNING

A Dissertation Presented

by

ANDERS JONSSON

Approved as to style and content by:

____

Andrew G. Barto, Chair

____

Sridhar Mahadevan, Member

____

Shlomo Zilberstein, Member

____

Neil E. Berthier, Member

____

W. Bruce Croft, Department Chair
Department of Computer Science

To my family.

ACKNOWLEDGEMENTS

In December 2002 I travelled to Vancouver, Canada, with my sister. When our grandmother was a child, she and her family emigrated from Sweden to Canada and settled near a small town called Hope in the vicinity of Vancouver. After the depression struck in 1929, the financial prospects were poor, so the family decided to return to Sweden. However, several of the Swedes who came with our grandmother stayed in British Columbia, and in 2002, she still had friends in Vancouver. They were very welcoming towards their former countrymen and invited me and my sister to stay in their home. They treated us as if we were part of the family and gave us extensive tours of the city and its surroundings.

During our second day in Vancouver, we drove to see the place where our grandmother had lived. After living for several years in the U.S., it felt special to experience the only connection my family has ever had with North America. Our grandmother's friends grew up in Hope and were telling us stories about their childhood. As we were driving through the small town, one of them pointed at a weirdly shaped tree. "That tree is called the H-Tree, since the branches of two trees have grown together to form the horizontal line of the H. We used to play there when we were children."

Two years prior, I had worked on the H-Tree algorithm that is part of this dissertation. I am not a superstitious person, but something struck within me when I heard about the H-Tree in the place where my grandmother lived as a child. The previous year I had taken a sabbatical from my graduate studies and travelled to Chile to lead hiking trips for young adults. During my sabbatical, I was evaluating my career choices and putting into doubt whether I wanted to continue doing research. I believe

that this day in Hope, for the first time I felt certain that I had made the right choice when coming to UMass to study for my Ph.D.

I would like to thank my parents for always being so supportive of me and for always letting me make my own decisions. I would also like to thank my sister, who I look up to and admire a lot. My wife, Rossana, has provided endless support and encouragement when I have struggled to find motivation, and has restored my harmony and confidence on more than one occasion when it was lacking.

The people in the Autonomous Learning Laboratory have also contributed in a big way to my Ph.D. First and foremost, I would like to thank my advisor, Andrew Barto, for being patient enough to have me as a graduate student. I would also like to thank Sridhar Mahadevan for valuable help and support on several occasions. I have had many fruitful discussions with Doina Precup, Ted Perkins, Balaraman Ravindran, Michael Rosenstein, Mohammad Ghavamzadeh, Khashayar Rohanimanesh, Özgür Şimşek, and Alicia "Pippin" Wolfe. The other people in the lab have contributed to making it a fun and inspiring environment to work in: Sascha Engelbrecht, Matt Schlesinger, Michael Kositsky, Amy McGovern, Ashvin Shah, Sarah Osentoski, Victoria Manfredi, Andrew Stout, Colin Barringer, Jeff Johns, Chris Vigorito, George Konidaris, and Kimberly Ferguson.

Finally, I have enjoyed the fortune of spending my time with many good friends outside of research. My fellow players on the soccer field with whom I shared many hours of sweat and toil. The people in the Vocal Jazz Ensemble who taught me so much about music. The students in the Graduate Employee Organization who never stop working for what they believe in. Finally, I feel especially grateful to my closest friends who have made my stay in Amherst such a pleasant one, whether at barbecues or during volleyball games or at the movies. I will always remember our time together with a smile.

ABSTRACT

A CAUSAL APPROACH TO HIERARCHICAL DECOMPOSITION IN REINFORCEMENT LEARNING

FEBRUARY 2006

ANDERS JONSSON

M.Sc., ROYAL INSTITUTE OF TECHNOLOGY, STOCKHOLM

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Andrew G. Barto

Reinforcement learning provides a means for autonomous agents to improve their action selection strategies without the need for explicit training information provided by an informed instructor. Theoretical and empirical results indicate that reinforcement learning algorithms can efficiently determine optimal or approximately optimal policies in tasks of limited size. However, as the size of a task grows, reinforcement learning algorithms become less consistent and less efficient at determining a useful policy. A key challenge in reinforcement learning is to develop methods that facilitate scaling reinforcement learning algorithms up to larger, more realistic tasks.

We present a series of algorithms that take advantage of task structure to make reinforcement learning more efficient in realistic tasks that display such structure. In each algorithm, we assume that the state space of a task is factored, i.e., states are collections of values of a set of state variables. Our work combines hierarchical decomposition and state abstraction to reduce the size of a task prior to applying

reinforcement learning. Hierarchical decomposition breaks a task into several subtasks that can be solved separately. For hierarchical decomposition to simplify learning, it is critical that each subtask is easier to solve than the overall task. To achieve the goal of simplifying the subtasks, we perform state abstraction separately for each subtask.

We begin by presenting an algorithm that uses experience from the environment to dynamically perform state abstraction for each subtask in an existing hierarchy of subtasks. Since our goal is to automate hierarchical decomposition as well as state abstraction, a second algorithm uses a dynamic Bayesian network action representation to automatically decompose a task into a hierarchy of subtasks. In addition, the algorithm provides an efficient way to perform state abstraction for each resulting subtask. A third algorithm constructs compact representations of activities that represent solutions to the subtasks. These compact representations enable the use of planning to efficiently approximate solutions to higher-level subtasks without interacting with the environment. Our fourth and final algorithm provides a means to learn a dynamic Bayesian network representation of actions from experience in tasks for which the representation is not available prior to learning.

The dissertation provides a detailed description of each algorithm as well as some theoretical results. We also present empirical results of each algorithm in a series of experiments. In tasks that display certain types of structure, the simplifications introduced by our algorithms significantly improve the performance of reinforcement learning. The results indicate that our algorithms provide a promising approach to make reinforcement learning better suited to solve realistic tasks in which these types of structure are present.

CONTENTS

Page

ACKNOWLEDGEMENTS . . . . **v**

ABSTRACT . . . . **vii**

LIST OF TABLES . . . . **xii**

LIST OF FIGURES . . . . **xiii**

CHAPTER

1. INTRODUCTION **1**

1.1 Summary of the dissertation 3
1.2 Putting it together 8

2. BACKGROUND **10**

2.1 Markov decision processes 10
2.2 Reinforcement learning 12
2.3 Activities 13

2.3.1 Options 13

2.4 Bayesian networks 14
2.5 State abstraction 15

2.5.1 Partitions 15

2.6 DBN models of factored MDPs 16

3. OPTION-SPECIFIC STATE ABSTRACTION **18**

3.1 The U-Tree algorithm 19
3.2 H-Tree: Extending the U-Tree algorithm 22

3.2.1 Hierarchical memory 23

3.2.2 Intra-option state abstraction . . . 23
3.2.3 Experimental results . . . 24

3.3 Discussion . . . 30
3.4 Related work . . . 31

4. A CAUSAL APPROACH TO HIERARCHICAL DECOMPOSITION . . . 34

4.1 The VISA algorithm . . . 36

4.1.1 Causal graph . . . 36
4.1.2 Identifying exits . . . 38
4.1.3 Introducing options . . . 41
4.1.4 Initiation set . . . 42
4.1.5 Termination condition . . . 44
4.1.6 Policy . . . 44
4.1.7 State abstraction . . . 46
4.1.8 Task option . . . 51
4.1.9 Exit transformations . . . 52
4.1.10 Merging strongly connected components . . . 52
4.1.11 Summary of the algorithm . . . 53
4.1.12 Limitations of the algorithm . . . 54

4.2 Experimental results . . . 54
4.3 Discussion . . . 61
4.4 Related work . . . 62

5. CONSTRUCTING COMPACT OPTION MODELS . . . 66

5.1 Multi-time option models . . . 67
5.2 Options in factored MDPs . . . 69
5.3 Partitions . . . 72
5.4 Finding useful partitions . . . 76

5.4.1 Tree operations . . . 76
5.4.2 Constructing partitions for exit options . . . 78
5.4.3 Distribution irrelevance . . . 82
5.4.4 Summary of the algorithm . . . 83

5.5 DBN model for options . . . 83
5.6 Experimental results . . . 85
5.7 Discussion . . . 86
5.8 Related work . . . 87

6. LEARNING DBN MODELS OF FACTORED MDPS **88**

6.1 Learning the structure of Bayesian networks 90
6.2 Learning a DBN model of factored MDPs 91

6.2.1 Active learning 94
6.2.2 Summary of the algorithm 97

6.3 Results 97
6.4 Discussion 100
6.5 Related work 101

7. CONCLUSION **104**

7.1 Future work 107

APPENDIX: PROOF OF THEOREM 5.1.2 . . . . **109**

BIBLIOGRAPHY **115**

LIST OF TABLES

Table **Page**

3.1 Parameters in our implementation of the U-Tree algorithm 30

4.1 Exits identified in the coffee task 41

5.1 Complexity of computing a multi-time model for each exit option 84

LIST OF FIGURES

Figure		Page
1.1	The causal graph of the coffee task	4
2.1	The DBN for action GO in the coffee task	17
3.1	Illustration of the Taxi task	24
3.2	Comparison between intra-option and regular state abstraction	27
3.3	Learned U-Trees for different policies	28
3.4	Comparison between hierarchical learning, with and without state abstraction, and flat learning	29
4.1	The DBN for action GO in the coffee task	37
4.2	The causal graph of the coffee task	39
4.3	HEX-Q's state variable ordering in the coffee task	40
4.4	The transition graph (left) and reachability tree (right) of the strongly connected component containing $S_{\mathtt{U}}$	43
4.5	The hierarchy of options discovered by the VISA algorithm in the coffee task	51
4.6	Illustration of the AGV task	56
4.7	Results of learning in the coffee task	57
4.8	Results of learning in the Taxi task	58
4.9	Results of learning in the Factory task	59

4.10 Results of learning in the AGV task . . . 60

5.1 The tree $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}}$ and the restriction $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}} \mid (S_{\mathtt{R}} = R)$ . . . 77

5.2 The trees $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}}$, $\mathcal{T}_{\mathtt{U}}$, and the intersection $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}} \cap \mathcal{T}_{\mathtt{U}}$ . . . 78

5.3 The policy of the exit option associated with the exit $\langle (S_{\mathtt{L}} = L), \mathtt{BC} \rangle$ . . . 79

5.4 The tree $\mathcal{T}_{\pi}^{o} \cap \mathcal{T}_{\mathtt{W}}$ and the result of MakeSSP($\mathcal{T}_{\pi}^{o} \cap \mathcal{T}_{\mathtt{W}}$) . . . 80

5.5 DBN for the option associated with $\langle (S_{\mathtt{L}} = L), \mathtt{BC} \rangle$ . . . 84

5.6 Hierarchical vs. flat SPI in the Taxi task . . . 86

6.1 Intermediate configuration of the tree $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}}$ during learning . . . 92

6.2 Results of learning DBNs in the coffee task . . . 98

6.3 Results of learning DBNs in the Taxi task . . . 99

6.4 Results of learning DBNs in the AGV task . . . 100

CHAPTER 1

INTRODUCTION

A central research topic in artificial intelligence is the development of algorithms that solve sequential decision problems, which are problems that require repeated decisions in dynamic environments to achieve one or several criteria. Instances of sequential decision problems appear in such diverse fields as industrial manufacturing, search-and-rescue operations, Internet search, bioinformatics, electronic commerce, space travel, board games, etc. Several factors contribute to making sequential decision problems challenging. Reward may be delayed, which makes it difficult to assess which decisions contribute to the desired outcome (i.e., the credit assignment problem). The number of states is exponential in the number of variables describing a task, causing solution techniques to scale poorly to large tasks (i.e., the curse of dimensionality). There are infinite ways to model a sequential decision problem (i.e., the curse of modeling). The underlying state may be hidden and only partially observable to the decision maker.

Reinforcement learning (Sutton and Barto, 1998) is a family of algorithms for solving stochastic sequential decision problems, usually modeled as Markov decision processes, or MDPs (Bellman, 1957). These algorithms use dynamic programming-style updates to propagate reward through the state space or along trajectories. Given enough time, reinforcement learning can solve the credit assignment problem for MDPs, and several reinforcement learning algorithms are guaranteed to converge to optimal solutions. However, because of the curse of dimensionality, these algorithms become intractable as the number of variables describing a task grows. For

reinforcement learning to become more useful in practice and better suited to real-life applications, it is necessary to scale reinforcement learning algorithms to increasingly large tasks.

One approach to scaling reinforcement learning is through the use of function approximation, which involves estimating an optimal solution of an MDP using a set of parameters significantly smaller than the number of states. Reinforcement learning combined with function approximation has had considerable success in tasks such as backgammon (Tesauro, 1994), job-shop scheduling (Zhang and Dietterich, 1995), and elevator dispatching (Crites and Barto, 1996). However, researchers have not been able to repeat the success of reinforcement learning with function approximation in other realistic tasks. It seems as if current techniques for function approximation do not generalize well to arbitrary tasks.

Another approach to scaling reinforcement learning is to simplify tasks as much as possible prior to learning. Since reinforcement learning algorithms have been empirically and theoretically shown to work well in tasks with limited size, it makes sense to reduce the size of a task before applying these learning algorithms. Realistic tasks usually display several forms of structure that make it possible to simplify them. Exploiting structure can bring the size of a task down to a reasonable level that gives reinforcement learning algorithms a better chance of finding a good approximate solution. In this dissertation, we will take this second approach to scaling reinforcement learning.

We will focus on two types of simplification: hierarchical decomposition and state abstraction. Hierarchical decomposition divides a task into a hierarchy of activities (also known as macro-actions or temporally-extended actions), which represent stand-alone subtasks that can be solved independently. If each subtask is significantly easier to solve than the overall task, hierarchical decomposition can cause an important reduction in complexity. There exist three models of activities in reinforcement

learning: Hierarchical Abstract Machines, or HAMs (Parr and Russell, 1998), options (Sutton et al., 1999), and MAXQ (Dietterich, 2000a). State abstraction ignores part of the information inherent in the state description, effectively reducing the size of the state space. Under certain conditions, state abstraction preserves optimality of MDPs (Dean and Givan, 1997; Ravindran, 2004).

The motivation for this dissertation is to develop novel algorithms that simplify sequential decision tasks prior to learning. The hope is that the simplification will increase the probability of reinforcement learning algorithms to successfully approximate good solutions. We want our algorithms to be efficient, robust, and generalize to a significant number of tasks. If possible, the algorithms should ameliorate the curse of dimensionality and not rely on quantities proportional to the size of the state space. We will focus on tasks that have factored state spaces, i.e., tasks whose states are described by the values of a collection of state variables. We believe that most realistic tasks fall within this category.

1.1 Summary of the dissertation

As a first step, we develop an algorithm, called the H-Tree algorithm, that performs state abstraction from experience in an existing hierarchy of options (Sutton et al., 1999), a special case of activities. Our algorithm is a modification of the U-Tree algorithm (McCallum, 1995), which gathers data in the form of transition instances and performs state abstraction in partially observable Markov decision processes, or POMDPs (Åström, 1965). We modify McCallum's definition of a transition instance to include options, and perform state abstraction separately for each option. Results indicate that option-specific state abstraction significantly accelerates learning in a hierarchy of options. We also introduce the idea of intra-option state abstraction: using experience from the execution of one option to perform state abstraction for

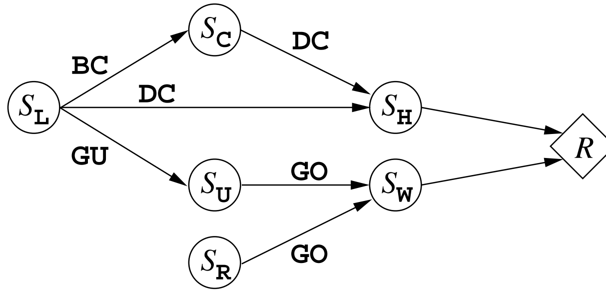

Figure 1.1. The causal graph of the coffee task

other options. Our experiments show that experience accumulates more quickly and more accurately using intra-option state abstraction.

Our goal is to automate hierarchical decomposition as well as state abstraction. We present the Variable Influence Structure Analysis, or VISA, algorithm, which uses causal relationships between variables to perform hierarchical decomposition of factored MDPs. The VISA algorithm is based on the assumption that the transition probabilities and expected reward of factored MDPs are modeled using dynamic Bayesian networks, or DBNs (Dean and Kanazawa, 1989). The DBN model captures conditional independencies between the state variables describing a factored MDP. The VISA algorithm uses the DBNs to construct a causal graph which determines causal relationships between state variables. VISA introduces options that cause the values of state variables to change. At the top level, the algorithm introduces a task option that corresponds to the original task. The causal relationships enable efficient state abstraction for each option, effectively reducing the size of the original task. The VISA algorithm is useful in tasks for which the values of key state variables change relatively infrequently.

Figure 1.1 illustrates the causal graph of the coffee task (Boutilier et al., 1995), which we will describe in detail in a later chapter. The causal graph has one node for each state variable, plus one node corresponding to expected reward. Each edge in the graph indicates that there are one or several causal relationships between two

state variables, conditional on actions that appear as labels on the edge. The VISA algorithm uses the causal graph to perform state abstraction separately for each option. An option that causes the value of a state variable S_i to change only needs to distinguish between values of state variables that have edges to S_i in the causal graph. This type of option-specific state abstraction significantly reduces the complexity of computing an option policy. If the causal graph contains cycles, the VISA algorithm gets rid of cycles by computing the strongly connected components of the graph.

The HEXQ algorithm (Hengst, 2002) determines causal relationships between state variable by counting the frequency with which the values of state variables change. HEXQ identifies exits, i.e., pairs of state variable values and actions that cause the values of state variables to change. The VISA algorithm also identifies exits, but unlike HEXQ, VISA uses the causal graph to determine causal relationships between state variables. Since the causal graph captures causal relationships more realistically than the frequency of change heuristic, VISA can decompose more general tasks than can HEXQ. The VISA algorithm introduces one option for each exit, and uses sophisticated techniques to determine the components of each option.

The VISA algorithm exploits sparse conditional dependence between state variables such that the causal graph contains two or more strongly connected components. In addition, the algorithm is more efficient when changes in the values of state variables occur relatively infrequently. How likely is a realistic task to display this type of structure? It is impossible for us to determine the percentage of tasks in which the aforementioned structure is present. However, our intuition tells us that it is not uncommon to encounter tasks that display this structure.

For example, consider any robot navigation task in which the robot has to pay attention to objects. In many instances, the location of the robot is independent of the state of these objects. However, the robot can affect the state of the objects if it is in the correct location. In this case, the causal graph will contain one strongly

connected component containing the state variable describing location, and other strongly connected components containing state variables describing the state of these objects. As a consequence, it is possible for the VISA algorithm to decompose the task. In Chapter 4, we performed experiments with the VISA algorithm in the Factory task, which does not involve navigation, so the structure is not limited to navigation tasks.

The DBN model of a factored MDP contains one DBN for each action, compactly describing the transition probabilities and expected reward associated with the action. Several researchers have developed efficient algorithms for solving factored MDPs when a DBN model is available (Boutilier et al., 1995; Feng et al., 2003; Guestrin et al., 2001; Jonsson and Barto, 2005; Kearns and Koller, 1999). If a similar compact DBN model were available for each option, these algorithms could solve factored MDPs decomposed into hierarchies of options. Existing techniques for constructing option models require enumeration of the state space, which scales poorly to large tasks. We develop an algorithm for constructing compact DBN models of options that does not require enumeration of the state space. The DBN model enhances the description of a learned option and makes it possible to learn and plan with options using the more efficient algorithms that use DBN models to solve MDPs.

Our algorithm for constructing compact option models makes several contributions. First, we analyze the complexity of constructing a DBN model that makes it possible to treat a learned option as a single unit during learning and planning. We investigate how to reduce the complexity through the use of partitions with certain properties. Finally, we show how to construct partitions with the required properties for a particular class of tasks when a compact DBN model of primitive actions is given. To construct partitions, we develop novel operations on decision trees. Results indicate that our technique can significantly reduce the complexity of constructing compact option models.

It is unrealistic to assume that a DBN model of a factored MDP is always available prior to learning. We address the problem of learning DBN models from experience. There exist algorithms in the literature for learning the structure of Bayesian networks (Buntine, 1991; Friedman et al., 1998; Heckerman et al., 1995). However, these algorithms assume that a data set is given, whereas solution techniques for MDPs typically have to gather data in the form of transitions and reward through interaction with the environment. The complexity of learning DBNs heavily depends on the time it takes to collect data. It is possible to accelerate data collection by selecting high-quality data instances through a process called active learning. Researchers have developed techniques for active learning of Bayesian networks (Murphy, 2001; Steck and Jaakkola, 2002; Tong and Koller, 2001). These techniques perform experiments by clamping a subset of the variables to fixed values and sampling over the remaining variables.

We assume that it is only possible to sample MDPs along trajectories, not in arbitrary states. The only way to gather information about transitions and reward is by executing an action in the current state. Since it is not possible to simulate the effect of actions in hypothetical states, we cannot perform experiments by clamping a subset of the variables to fixed values. Consequently, we cannot apply existing techniques for active learning. However, there is still an opportunity to perform active learning of DBNs in factored MDPs. Because the DBN model of a factored MDP consists of one DBN for each action, by selecting an action we effectively select a DBN for which to collect data. It is thus possible to consider policies for action selection whose utility lie in efficient data collection.

We develop an algorithm for learning DBNs that grows trees representing the conditional probabilities of the DBNs. Our algorithm collects data instances by executing actions and grows the trees as soon as a minimum number of data instances correspond to each relevant value of each split variable. The algorithm uses the Bayesian

Information Criterion (BIC) (Schwartz, 1978) and the likelihood-equivalent Bayesian Dirichlet metric (BDe) (Heckerman et al., 1995) to evaluate potential refinements. We assume that no data is available to begin with and develop a technique for active learning of DBNs to accelerate data collection. The time to collect data is minimized if the distribution of data instances across values of each potential split variable is perfectly uniform. We use the entropy of the distributions to measure uniformity and select actions that maximize the total entropy of the distributions.

In each chapter, we illustrate the utility of one of our algorithms by conducting a series of empirical experiments. Whenever possible, we compare our algorithm to existing, state-of-the-art algorithms with similar application. We present the results of the experiments in graphs that illustrate the convergence time and utility level of each algorithm. In some cases, we also present theoretical results regarding the complexity of our algorithms.

1.2 Putting it together

We present our algorithms in four separate chapters, but each algorithm could be viewed as part of a larger system. Given a factored MDP, such a system would perform the following steps:

1. Learn a DBN model of the factored MDP (Chapter 6)
2. Perform hierarchical decomposition using the DBN model (Chapter 4)
3. Perform state abstraction for each resulting option (Chapter 4)
4. Refine the state abstraction for each option (Chapter 3)
5. Construct a compact representation of each option (Chapter 5)
6. Learn the policy of each option, including task option (Chapter 4)

The reason that the chapters do not appear in order is that we present the algorithms in chronological order of development. How efficient would this system be at solving factored MDPs? Experimental results in Chapter 4 indicate that steps 2, 3, and 6 above can be implemented very efficiently in factored MDPs that display the structure exploited by the VISA algorithm. Steps 4 and 5 can contribute to making step 6 more efficient. Consequently, we believe that step 1 currently represents the main limitation of this system. Our approach to learning DBN models of factored MDPs is only a first step, and more work is needed in this area to make the system as a whole more efficient.

CHAPTER 2

BACKGROUND

2.1 Markov decision processes

A Markov decision process, or MDP (Bellman, 1957), is a model of a stochastic sequential decision problem. Formally, a finite MDP is a tuple $\mathcal{M} = \langle S, A, \Psi, P, R \rangle$, where S is a finite set of states, A is a finite set of actions, $\Psi \subseteq S \times A$ is a set of admissible state-action pairs, P is a transition probability function, and R is an expected reward function. Let $A_s \equiv \{a' \in A \mid (s, a') \in \Psi\}$ be the set of admissible actions in state $s \in S$. Ψ is such that for each state $s \in S$, the set of admissible actions A_s is non-empty, i.e., there is at least one admissible action. As a result of executing action $a \in A_s$ in state $s \in S$, the process transitions to state $s' \in S$ with probability $P(s' \mid s, a)$ and provides the decision maker with an expected reward $R(s, a)$. P is such that for each admissible state-action pair $(s, a) \in \Psi$, $\sum_{s' \in S} P(s' \mid s, a) = 1$. The transition probability function P and expected reward function R constitute the dynamics of MDP $\mathcal{M}$.

A stochastic policy π selects action $a \in A_s$ in state $s \in S$ with probability $\pi(s, a)$. π is such that for each state $s \in S$, $\sum_{a \in A_s} \pi(s, a) = 1$. In the discounted case, the value function V^π of policy π is the solution to the set of linear equations

$$V^\pi(s) = \sum_{a \in A_s} \pi(s, a) \left[R(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) V^\pi(s') \right], \qquad (2.1)$$

where $\gamma \in (0, 1]$ is a discount factor. If we view V^π as a vector, we can define an operator $\mathcal{T}^\pi$ and write the equations in (2.1) as $V^\pi = \mathcal{T}^\pi V^\pi$. The optimal value

function V^* is the solution to the Bellman optimality equation (Bellman, 1956):

$$V^*(s) = \max_{a \in A_s} \left[R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V^*(s') \right]. \tag{2.2}$$

Equivalently, we define an operator $\mathcal{T}^*$ to write the Bellman equation as $V^* = \mathcal{T}^* V^*$. An optimal policy π^* is any policy whose value function is V^*, i.e., in each state $s \in S$ it only assigns positive probabilities to actions in the set

$$A^*(s) = \arg\max_{a \in A_s} \left[R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V^*(s') \right]. \tag{2.3}$$

We define an operator $\mathcal{A}^*$ such that $\pi^* = \mathcal{A}^* V^*$ is an optimal policy.

If the dynamics of an MDP are known, it is possible to use dynamic programming to compute an optimal policy. One such dynamic programming algorithm is value iteration, which maintains an estimate V^t of the optimal value function V^*. The algorithm successively adjusts its value estimate by performing the update $V^{t+1} = \mathcal{T}^* V^t$. The resulting policy is given by $\pi = \mathcal{A}^* V^t$. Policy iteration (Howard, 1960) maintains a current policy π^t and an estimate V^{π^t} of the policy's value function. The algorithm alternates between value estimation and policy improvement. In the value estimation step, the algorithm uses dynamic programming to compute the solution to the equations $V^{\pi^t} = \mathcal{T}^{\pi^t} V^{\pi^t}$. Thereafter, the algorithm computes an improved policy $\pi^{t+1} = \mathcal{A}^* V^{\pi^t}$. Both value iteration and policy iteration are guaranteed to converge to an optimal policy (Puterman and Brumelle, 1979; Puterman, 1990).

A partially observable Markov decision process, or POMDP (Åström, 1965), models a stochastic sequential decision process in which the underlying state cannot be directly observed. Formally, a finite POMDP is a tuple $\mathcal{P} = \langle S, \Omega, \phi, A, \Phi, P, R \rangle$, where Ω is a finite set of observations, ϕ is an observation probability function, and $\Phi \subseteq \Omega \times A$ is a set of admissible observation-action pairs. The current state $s \in S$ of a POMDP is unobservable. Instead, the decision maker makes an observation $\omega \in \Omega$

with probability $\phi(s, \omega)$. ϕ is defined such that for each state $s \in S$, $\sum_{\omega \in \Omega} \phi(s, \omega) = 1$. Let $A_\omega \equiv \{a' \in A \mid (\omega, a') \in \Phi\}$ be the set of admissible actions for observation $\omega \in \Omega$. Φ is defined such that for each observation $\omega \in \Omega$, the set of admissible actions A_ω is non-empty, i.e., there is at least one admissible action for each observation.

2.2 Reinforcement learning

Reinforcement learning (Sutton and Barto, 1998) is a family of algorithms that aim at computing an optimal or near-optimal policy of an MDP. Reinforcement learning is especially useful when the dynamics of an MDP are unknown, in which case it is not possible to apply dynamic programming to compute an optimal policy. There exist several reinforcement learning algorithms that approximate an optimal policy through interaction with the environment. Value-based reinforcement learning algorithms maintain an estimate V of the optimal value function V^*, or an estimate Q of the optimal action-value function Q^*, defined as

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) V^*(s'). \tag{2.4}$$

The optimal policy is implicitly defined in each state as the action or actions which result in the highest expected value. All reinforcement learning algorithms act in the environment by executing actions according to some exploration scheme and recording ensuing transitions and rewards. Following each transition, value-based reinforcement learning algorithms adjust the local value estimate for the previous state or state-action pair according to one of several update rules, possibly causing the current policy to change. Examples of update rules include $TD(\lambda)$ (Sutton, 1988), Q-learning (Watkins, 1989), and SARSA (Sutton, 1996). If each state-action pair is visited infinitely often, if value estimates are stored in a look-up table, and if the step size parameter is appropriately reduced, reinforcement learning algorithms are guaranteed to find an optimal policy (Sutton, 1988; Watkins and Dayan, 1992).

2.3 Activities

An activity (also known as a macro-action or a temporally-extended action) is a closed-loop policy that takes multiple time steps to execute. Activities exploit recurrence by representing subroutines that are executed multiple times during successful solution of a task. For example, if the task is to collect balls and put them into a bin, the subroutine of putting a ball into the bin will be executed multiple times. If an activity has been learned in one task, it can be reused in a second task which requires execution of the same subroutine. Activities also enable more efficient exploration by providing high-level behavior that looks ahead to the completion of the subroutines. Finally, activities can accelerate learning if the subroutines are easier to learn than the overall task. There exist three models of activities in reinforcement learning: Hierarchical Abstract Machines, or HAMs (Parr and Russell, 1998), options (Sutton et al., 1999), and MAXQ (Dietterich, 2000a).

A semi-Markov decision process, or SMDP (Puterman, 1994), is a model of stochastic sequential decision problems in which actions can take variable amounts of time. One way to form an SMDP is by adding activities to the action set of an MDP. The policy of an SMDP selects between activities, which in turn select between actions, resulting in a hierarchical representation of the task. Several reinforcement learning algorithms have been extended to SMDPs (Bradtke and Duff, 1995; Sutton et al., 1999; Dietterich, 2000a).

2.3.1 Options

An option (Sutton et al., 1999) is a model of an activity. Given an MDP $\mathcal{M} = \langle S, A, \Psi, P, R \rangle$, an option is a tuple $o = \langle I, \pi, \beta \rangle$, where $I \subseteq S$ is an initiation set, π is a policy, and β is a termination function. Option o can be executed in any state $s \in I$, repeatedly selects actions $a \in A$ according to π, and terminates in state $s' \in S$ with probability $\beta(s')$. An action $a \in A$ can be viewed as an option with initiation

set $I = \{s \in S \mid (s, a) \in \Psi\}$ whose policy always selects a and that terminates in all states with probability 1.

Ravindran (2004) showed that an option o can be constructed by solving the stand-alone task given by the option SMDP $\mathcal{M}_o = \langle S_o, O_o, \Psi_o, P_o, R_o \rangle$, where $S_o \subseteq S$ is the subset of states for which option o is defined and O_o is a set of options. The set of admissible state-option pairs $\Psi_o \subseteq S_o \times O_o$ is determined by the initiation sets of options in O_o. The transition probability function P_o is determined by the transition probability function P of the underlying MDP and the policies and termination functions of the options in O_o. The expected reward function R_o is independent of the expected reward function R of the underlying MDP and can be selected to reflect the desired behavior of option o. The policy π of option o can be defined as the optimal policy of the option SMDP $\mathcal{M}_o$.

2.4 Bayesian networks

Let $\mathbf{X}$ be a set of discrete variables, and let $\mathbf{x}$ be an assignment of values to the variables in $\mathbf{X}$. Let $f_{\mathbf{Y}}$, $\mathbf{Y} \subseteq \mathbf{X}$, be a projection such that if $\mathbf{x}$ is an assignment to $\mathbf{X}$, $f_{\mathbf{Y}}(\mathbf{x})$ is $\mathbf{x}$'s assignment to $\mathbf{Y}$. A Bayesian network (Pearl, 1988) is a tuple $B = \langle G, \theta \rangle$, where G is a directed acyclic graph with one node per variable $X_i \in \mathbf{X}$, and θ is a set of parameters defining the conditional probabilities of the variables. The joint probability distribution of the variables is given by

$$P(\mathbf{x}) = \prod_i P(X_i = f_{\{X_i\}}(\mathbf{x}) \mid \mathbf{Pa}(X_i) = f_{\mathbf{Pa}(X_i)}(\mathbf{x})), \tag{2.5}$$

where $\mathbf{Pa}(X_i) \subseteq \mathbf{X}$ is the subset of variables with edges to X_i in G and the probabilities $P(X_i = f_{\{X_i\}}(\mathbf{x}) \mid \mathbf{Pa}(X_i) = f_{\mathbf{Pa}(X_i)}(\mathbf{x}))$ are defined by parameters in θ.

A dynamic Bayesian network, or DBN (Dean and Kanazawa, 1989), is a Bayesian network that models the evolution of a set of variables in a temporal process. The

directed acyclic graph of a DBN has two layers of nodes: one layer representing the current values of the variables, and one layer representing the next values of the variables. The edges between layers are unidirectional and always point from the current layer to the next layer. There can also be edges between nodes within a layer.

2.5 State abstraction

Each state of an MDP contains information about the current configuration of the environment. Usually, not all information contained in a state is relevant to selecting an optimal action. State abstraction is the process of ignoring part of the information contained in a state in the hope of focusing on the information that is relevant to selecting an optimal action. As a result of state abstraction, there are less distinct situations to consider, simplifying learning of the MDP.

2.5.1 Partitions

A partition Λ of the set of states S of an MDP is a collection of disjoint subsets, or blocks, $\lambda \subseteq S$ such that $\bigcup_{\lambda \in \Lambda} \lambda = S$. $[s]_\Lambda \in \Lambda$ denotes the block to which state $s \in S$ belongs. A function $f : S \to X$ from S to an arbitrary domain X induces a partition Λ_f of S such that for each pair of states $(s_i, s_j) \in S^2$, $[s_i]_{\Lambda_f} = [s_j]_{\Lambda_f}$ if and only if $f(s_i) = f(s_j)$. Let Λ_1 and Λ_2 be two partitions of S. Partition Λ_1 refines Λ_2, denoted $\Lambda_1 \leq \Lambda_2$, if and only if, for each pair of states $(s_i, s_j) \in S^2$, $[s_i]_{\Lambda_1} = [s_j]_{\Lambda_1}$ implies that $[s_i]_{\Lambda_2} = [s_j]_{\Lambda_2}$. The relation $\leq$ is a partial ordering on the set of partitions of S.

Dean and Givan (1997) and Ravindran (2004) defined two properties of partitions with respect to an MDP $\mathcal{M}$. A partition Λ has the stochastic substitution property if, for each pair of states $(s_i, s_j) \in S^2$, each action $a \in A$ and each block $\lambda \in \Lambda$, $[s_i]_\Lambda = [s_j]_\Lambda$ implies that $\sum_{s_k \in \lambda} P(s_k \mid s_i, a) = \sum_{s_k \in \lambda} P(s_k \mid s_j, a)$. Λ is reward respecting if for each pair of states $(s_i, s_j) \in S^2$ and each action $a \in A$, $[s_i]_\Lambda = [s_j]_\Lambda$ implies that $R(s_i, a) = R(s_j, a)$. A partition Λ that has the stochastic substitution

property and is reward respecting partition induces a reduced MDP which has fewer states and preserves optimality (Dean and Givan, 1997; Ravindran, 2004).

2.6 DBN models of factored MDPs

A factored MDP is described by a set of discrete state variables $\mathbf{S}$. Each state variable $S_i \in \mathbf{S}$ takes on a value in the set $Val(S_i)$. The set of states $S \subseteq \times_i Val(S_i)$ is a subset of the Cartesian product of the value sets of all state variables. A state $\mathbf{s} \in S$ assigns a value $f_{\{S_i\}}(\mathbf{s}) \in Val(S_i)$ to each state variable $S_i \in \mathbf{S}$. We define a context $\mathbf{c}$ as an assignment to a subset of the state variables $\mathbf{C} \subseteq \mathbf{S}$, i.e., a partial assignment of values to the state variables.

We illustrate factored MDPs using the coffee task (Boutilier et al., 1995), in which a robot has to deliver coffee to its user. The coffee task is described by six binary state variables: $S_{\mathtt{L}}$, the robot's location (office or coffee shop); $S_{\mathtt{U}}$, whether the robot has an umbrella; $S_{\mathtt{R}}$, whether it is raining; $S_{\mathtt{W}}$, whether the robot is wet; $S_{\mathtt{C}}$, whether the robot has coffee; and $S_{\mathtt{H}}$, whether the user has coffee. The robot has four actions: `GO`, causing its location to change and the robot to get wet if it is raining and it does not have an umbrella; `BC` (buy coffee) causing it to hold coffee if it is in the coffee shop; `GU` (get umbrella) causing it to hold an umbrella if it is in the office; and `DC` (deliver coffee) causing the user to hold coffee if the robot has coffee and is in the office. All actions have a chance of failing. The robot gets a reward of 0.9 whenever the user has coffee plus a reward of 0.1 whenever it is dry.

Boutilier et al. (1995) developed a compact model of factored MDPs that uses DBNs to represent the effect of actions. The DBN model can be viewed as a probabilistic version of the STRIPS action formulation (Fikes and Nilsson, 1971), which is widely used in planning algorithms that solve deterministic sequential decision problems, since both describe the cause-effect relationships of actions. The DBN model contains one DBN for each action $a \in A$ of a factored MDP. Figure 2.1 illustrates the

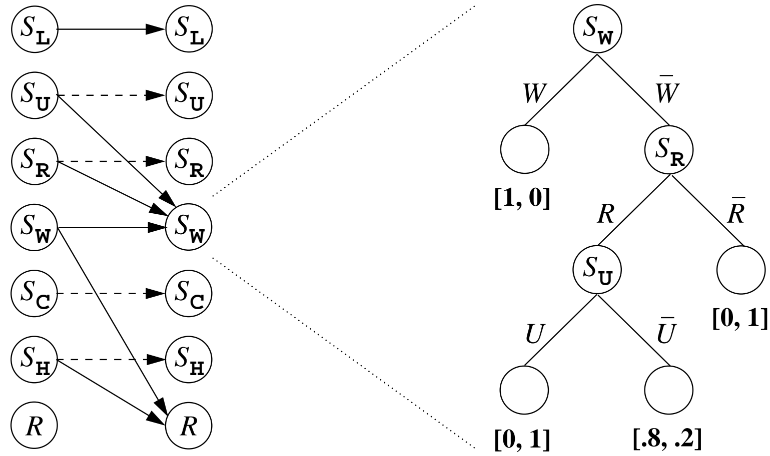

Figure 2.1. The DBN for action `GO` in the coffee task

DBN for action `GO` in the coffee task. There are two nodes for each state variable and two nodes corresponding to expected reward. Nodes on the left represent the values of variables prior to executing `GO`, and nodes on the right represent the values after executing `GO`. The value of a state variable S_i as a result of executing `GO` depends on the values of state variables that have edges to S_i in the DBN. A dashed line indicates that a state variable is unaffected by `GO`.

Each state variable S_i in the DBN for action a has an associated conditional probability distribution, determining the resulting value of S_i after executing a. We assume that conditional probabilities are stored in trees. Figure 2.1 illustrates the conditional probability tree associated with state variable $S_{\texttt{W}}$ and action `GO`. For example, if the robot is dry ($\overline{W}$), it is raining (R), and the robot does not have an umbrella ($\overline{U}$), the robot becomes wet with probability 0.8 as a result of executing `GO`. We assume that there are no edges between state variables in a layer of the DBN. In this case, the DBN model cannot represent arbitrary transition probabilities. Instead, the transition probabilities are approximated according to $P(\mathbf{s}' \mid \mathbf{s}, a) \approx \prod_{S_i \in \mathbf{S}} P_i(S_i = f_{\{S_i\}}(\mathbf{s}') \mid \mathbf{Pa}(S_i) = f_{\mathbf{Pa}(S_i)}(\mathbf{s}), a)$, where P_i is the conditional probability distribution of state variable S_i represented by the DBN for a.

CHAPTER 3

OPTION-SPECIFIC STATE ABSTRACTION

Research on activities has made it possible to exploit the hierarchical structure of a task by representing recurring subroutines that are repeatedly executed during solution of the task. An activity that has been learned in one task can be reused in another task that requires execution of the same subroutine. Activities also enable more efficient exploration of a novel task since they provide a learning agent with larger chunks of behavior that enable the agent to look ahead multiple time steps and perform search at a higher level. The ability to perform hierarchical decomposition has enriched the reinforcement learning framework and increased the possibility of exploiting task structure.

Most research on activities has focused on detecting recurring subroutines. Although this is clearly an essential step in the process of hierarchical decomposition, it is usually necessary to complement this process with state abstraction. If the stand-alone task associated with an activity is equally difficult to solve as the overall task, the complexity of solving the task is increased with each additional activity. To truly benefit from hierarchical decomposition, the stand-alone task associated with each activity should be easier to solve than the overall task. State abstraction can reduce the complexity of solving the stand-alone tasks of activities by ignoring irrelevant information.

Usually, different information is relevant for different activities. For example, if I perform the task of walking to the door in a room, the locations of objects in the room are relevant, but not the color of the door. On the other hand, if my task is to paint

the door, the color of the door is important, but not the location of objects in the room (unless they obstruct the door). Which information it is safe to ignore depends on the activity currently executing. To maximize the gain of state abstraction, we should perform state abstraction separately for each activity.

In this chapter, we develop an algorithm that uses experience from interaction with the environment to perform state abstraction separately for each option (Sutton et al., 1999), a model of activities. We assume that there is an existing hierarchy of options, and focus solely on the process of performing state abstraction. Our algorithm is an extension of the U-Tree algorithm (McCallum, 1995), which performs state abstraction in POMDPs by gathering transition instances through interaction with the environment. We modify the definition of a transition instance to include activities.

3.1 The U-Tree algorithm

The U-Tree algorithm (McCallum, 1995) forms a tree representation of a POMDP $\mathcal{P} = \langle S, \Omega, \phi, A, \Phi, P, R \rangle$ from experience with the environment. The representation makes it possible to perform state abstraction, simplifying the POMDP. The algorithm assumes that the set of observations Ω is factored, i.e., that a set of observation variables $\mathbf{\Omega}$ describes the current observation. Each observation ω is an assignment of values to the variables in $\mathbf{\Omega}$. The U-Tree algorithm executes actions in the environment according to an exploration scheme and records ensuing transition instances. A transition instance is a tuple $T_t =< T_{t-1}, a_{t-1}, r_t, \omega_t >$, where ω_t is the observation at time t, $a_{t-1} \in A$ is the previous action executed, r_t is the reward received during the transition to ω_t, and T_{t-1} is the previous transition instance. At the onset of learning, i.e., at time $t = 0$, T_0 is defined as an empty tuple.

The U-Tree algorithm maintains a tree, called the U-Tree, that sorts transition instances based on their components. Since the true state is hidden, the U-Tree

algorithm considers distinctions over actions and observation variables during the last H time steps, where H is a history index. In other words, the U-Tree induces a partition of the set $(A \times \Omega)^H$. A transition instance T_t recursively defines a history $h_t = (a_{t-H}, \omega_{t-H+1}, \ldots, a_{t-1}, \omega_t) \in (A \times \Omega)^H$ of actions and observations during the last H time steps. Based on its history h_t, each transition instance T_t maps to a unique leaf $L(T_t)$ of the U-Tree, at which it is stored.

The U-Tree algorithm performs state abstraction in the original POMDP by treating each leaf l of the U-Tree as a single state of an MDP. For each leaf-action pair (l, a) and each leaf l', the algorithm maintains an estimate $\hat{P}(l' \mid l, a)$ of the probability of transitioning from leaf l to leaf l' as a result of executing a. For each leaf-action pair (l, a), the algorithm also maintains an estimate $\hat{R}(l, a)$ of the expected reward as a result of executing a in leaf l. The estimated transition probabilities $\hat{P}$ and expected reward $\hat{R}$ can be computed from the transition instances at a leaf. Let $T(l, a, l')$ be the set of all recorded transition instances from leaf l to leaf l' via action a, i.e.,

$$T(l, a, l') = \{T_t \mid L(T_{t-1}) = l, a_{t-1} = a, \text{ and } L(T_t) = l'\}. \tag{3.1}$$

If no transition instances have been recorded as a result of executing action a in leaf l, $\hat{P}(l' \mid l, a)$ and $\hat{R}(l, a)$ take on predefined values. Otherwise, $\hat{P}(l' \mid l, a)$ and $\hat{R}(l, a)$ are computed as follows:

$$\hat{P}(l' \mid l, a) = \frac{|T(l, a, l')|}{|\bigcup_{l''} T(l, a, l'')|}, \tag{3.2}$$

$$\hat{R}(l, a) = \frac{\sum_{l'} \sum_{T_t \in T(l,a,l')} r_t}{|\bigcup_{l''} T(l, a, l'')|}. \tag{3.3}$$

The U-Tree algorithm performs reinforcement learning by maintaining an estimate $Q(l, a)$ of the optimal action-value of leaf-action pair (l, a). Following each time step, all action-values are updated using a full sweep of value iteration:

$$Q(l, a) = \hat{R}(l, a) + \gamma \sum_{l'} \hat{P}(l' \mid l, a) V(l'), \tag{3.4}$$

where $V(l') = \max_{a' \in A} Q(l', a')$ is the value of leaf l'.

The U-Tree algorithm starts out with a U-Tree containing a single root node. In other words, the algorithm ignores differences between any two situations, treating every situation the same. After every K steps, the algorithm performs a statistical test to determine whether or not to refine the U-Tree. At each leaf l, the algorithm considers as split variables each action or observation variable during the last H time steps not already refined. For each split variable at leaf l, the algorithm makes a temporary refinement. A refinement consists of introducing a fringe node l_k, a child of leaf l, for each value k of the split variable. The algorithm assumes a ranking on the variables and refines the U-Tree in order of rank. Each instance T_t that maps to l is distributed to the fringe nodes according to the value that T_t assigns to the split variable, and $\hat{P}$ and $\hat{R}$ are recomputed. The algorithm performs value iteration to estimate the action-value $Q(l_k, a)$ for each fringe node l_k and action $a \in A$.

The algorithm uses the Kolmogorov-Smirnov statistical test to compare the distribution of expected future discounted reward at leaf l with the distribution at each fringe node l_k. The distribution of expected future discounted reward at leaf l depends on the leaf's policy action a^*, given by $a^* = \arg\max_{a \in A} Q(l, a)$. The distribution at leaf l is composed of the expected future discounted reward $Q(T_t)$ associated with individual instances T_t consistent with leaf l and its policy action a^*. An instance T_t is consistent with l and a^* if $L(T_t) = l$ and $a_t = a^*$, and $Q(T_t)$ is given by

$$Q(T_t) = r_{t+1} + \gamma \sum_{l'} \hat{P}(l' \mid L(T_t), a_t) V(l'). \tag{3.5}$$

If the distibutions differ with sufficiently small significance, the temporary refinement is kept, and the fringe nodes l_k become new leaves of the U-Tree. This process continues until no more refinements are considered useful.

3.2 H-Tree: Extending the U-Tree algorithm

The H-Tree algorithm (Jonsson and Barto, 2001), where H-Tree is short for Hierarchical U-Tree, is an extension of the U-Tree algorithm to partially observable semi-Markov decision processes, or POSMDPs. POSMDPs are formed by adding activities to the action set A of a POMDP. The H-Tree algorithm uses options (Sutton et al., 1999) to model activities, so the action set A is replaced in the POSMDP by an option set O. The algorithm uses experience from the environment to perform option-specific state abstraction in an existing hierarchy of options.

We extended the U-Tree algorithm to POSMDPs by redefining the concept of a transition instance to include activities. Since we are dealing with POSMDPs, each option is associated with a stand-alone task given by the option POSMDP $\mathcal{P}_o = \langle S_o, \Omega_o, \phi_o, O_o, \Phi_o, P_o, R_o \rangle$. According to our new definition, a transition instance $T^o_{t_i} = \langle T^o_{t_{i-1}}, o_{t_{i-1}}, k_{t_i}, r_{t_i}, \omega_{t_i} \rangle$ is associated with an option o. Here, $T^o_{t_{i-1}}$ is the previous transition instance associated with o, $o_{t_{i-1}} \in O_o$ is the option previously executed by option o's policy, $k_{t_i} = t_i - t_{i-1}$ is the duration from the time option $o_{t_{i-1}}$ was initiated until it terminated, r_{t_i} is the sum of discounted reward received during the execution of $o_{t_{i-1}}$, and $\omega_{t_i} \in \Omega$ is the observation at time t_i.

The H-Tree algorithm maintains one U-Tree for each option $o \in O$. In addition, each option o maintains its own set of transition instances. The U-Tree of option o makes distinctions over the last H options and observations recorded during execution of option o. In other words, the U-Tree of option o induces a partition of the set $(O_o, \Omega)^H$. A transition instance $T^o_{t_i}$ recursively defines a history $h_{t_i} = (o_{t_{i-H}}, \omega_{t_{i-H+1}}, \ldots, o_{t_{i-1}}, \omega_{t_i}) \in (O_o \times \Omega)^H$ of the last H options and observations recorded during execution of o. Each transition instance $T^o_{t_i}$ of option o maps to a unique leaf $L(T^o_{t_i})$ of option o's U-Tree according to its history h_{t_i}.

The H-Tree algorithm grows the U-Trees in the same way as the U-Tree algorithm. The U-Tree of each option initially contains just one node. After each K time

steps, the H-Tree algorithm stops to consider refinements of the U-Trees. The algorithm uses SMDP Q-learning (Bradtke and Duff, 1995) to estimate the option-value $Q(l, o)$ of each leaf-option pair (l, o). Each refinement is evaluated by performing the Kolmogorov-Smirnov test on the distributions of expected future discounted reward at the leaf and fringe nodes. Refinements are kept if the distributions differ with sufficiently small statistical significance. The result is a collection of trees that performs state abstraction separately for each option.

3.2.1 Hierarchical memory

In addition to option-specific state abstraction, the H-Tree algorithm entails another benefit, namely hierarchical memory. A common criticism of the U-Tree algorithm is that it does not explain how to select the parameter H, the fixed history index used as a basis for action selection. Sometimes the algorithm needs to remember a key decision, regardless of whether it was made 10 or 100 time steps ago. The H-Tree algorithm does not provide a strategy for selecting H either. However, the hierarchical organization of POSMDPs makes it easier to remember important decision points. With our definition, the transition instance immediately preceding the current transition instance may have occured many time steps ago. We can safely assume that executing a subroutine is a key decision. Our algorithm may only need one or two transition instances to remember a key decision that occured many time steps ago, so the algorithm will need to look at much fewer previous transition instances when selecting an action. Memory is stored separately at each level in the hierarchy of options in the form of transition instances; hence the term hierarchical memory.

3.2.2 Intra-option state abstraction

Recall that the policy π of an option o selects between a set O_o of options at a lower level in the hierarchy. It is possible that the policies of several options share the same set of lower-level options. Since the option policies share the same set of

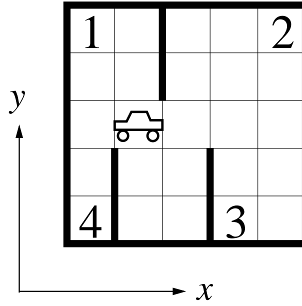

Figure 3.1. Illustration of the Taxi task

options, a lower-level option selected during execution of one option could have been selected during execution of any of the other options. Intra-option state abstraction means that when a transition instance is recorded during the execution of an option o, this instance is appended to all options whose policies select between the same set O_o of lower-level options. These other options treat the transition instance as if it had been recorded during their execution. This way, experience is accumulated faster and refinements can be made earlier and with higher confidence.

3.2.3 Experimental results

We tested the H-Tree algorithm on the Taxi task (Dietterich, 2000a), in which an agent – the taxi – moves around on a grid (Figure 3.1). The taxi is assigned the task of delivering passengers from their location to their destination, both chosen at random from the set of sites $Q = \{\mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}\}$. The taxi agent's observation $\omega = (x, y, l, d)$ is composed of the (x, y)-position of the taxi, the location $l \in Q \cup \{\mathtt{taxi}\}$ of the current passenger, and this passenger's destination $d \in Q$. The actions available to the taxi are `Pick-up`, `Drop-off`, and $\mathtt{Move}(m)$, $m \in \{\mathtt{N}, \mathtt{E}, \mathtt{S}, \mathtt{W}\}$, the four cardinal directions. With probability 0.2, a move action takes the taxi in a random direction. When a passenger is delivered, another passenger appears at a random site. The rewards provided to the taxi are:

19	for delivering a passenger
-11	for illegal `Pick-up` or `Drop-off`
-1	for any other action (including moving into walls)

Navigating to individual sites are recurring subroutines that precede each pick-up and delivery. For each site $q \in Q$, we introduced an option $\texttt{Navigate}(q) = \langle I_q, \pi_q, \beta_q \rangle$, where, letting Ω denote the set of observations and $\Omega_q = \{\omega \in \Omega \mid (x, y) = (x_q, y_q)\}$ the set of observations such that the taxi is at site q,

I_q : $\Omega - \Omega_q$

π_q : the policy for reaching site q that the agent needs to learn

β_q : 1 if ω is in Ω_q; 0 otherwise.

We further introduced a local reward R_q for $\texttt{Navigate}(q)$, identical to the global reward provided to the agent with the exception that $R_q = 9$ for reaching site q.

In our application of the H-Tree algorithm to the taxi task, each option remembered a maximum of 6,000 transition instances. If this length was exceeded, the oldest instance in the history was discarded. Expanding the tree was only considered if there were more than 3,000 instances. Since the H-Tree algorithm does not go back and reconsider refinements of the tree, it is important to reduce the number of incorrect refinements due to sparse statistical evidence. Therefore, our implementation only compared two distributions of expected future discounted reward if each contained more than 15 transition instances.

Because the taxi task is fully observable, we set the history index H of the H-tree algorithm to one. For exploration, we used an ϵ-softmax strategy, which picks a random action with probability ϵ and performs softmax exploration Sutton and Barto (1998) otherwise. Normally, tuning the softmax temperature τ provides a good balance between exploration and exploitation, but as the U-Trees evolve, a new

value of τ may improve performance. To avoid re-tuning τ, the ϵ-random part ensured that all actions were executed regularly.

We designed one set of experiments to examine the efficiency of intra-option state abstraction. We randomly selected one of the options `Navigate`(q) to execute, and randomly selected a new position for the taxi whenever it reached q, ignoring the issue of delivering a passenger. At the beginning of each learning run, we assigned a U-Tree containing a single node to each option. In one set of runs, the algorithm used intra-option state abstraction, and in another set, it used regular state abstraction in which the U-Trees of different options did not share any transition instances.

In a second set of experiments, the policies of the options and the overall Taxi task were learned in parallel. We allowed the policy of the overall task to choose between the options `Navigate`(q) and the actions `Pick-up` and `Drop-off`. The reward provided for the overall task was the sum of global reward and local reward of the option currently being executed (c.f. Digney, 1996). When a passenger was delivered, a new taxi position was selected randomly and a new passenger appeared at a randomly selected site.

The results from the experiments with intra-option state abstraction are shown in Figure 3.2. The graphs for intra-option state abstraction (solid) and regular state abstraction (dashed) are averaged over 5 independent runs. We tuned τ and ϵ for each set of learning runs to give maximum performance. At intervals of 500 time steps, the U-Trees of the options were saved and evaluated separately. The evaluation consisted of fixing a target, repeatedly navigating to that target for 25,000 time steps, randomly repositioning the taxi every time the target was reached, repeating for all targets, and adding the rewards. From these results, we conclude that (1) intra-option state abstraction converges faster than regular state abstraction, and (2) intra-option state abstraction achieves a higher level of performance. Faster convergence is due to the fact that the number of transition instances associated with the options increase

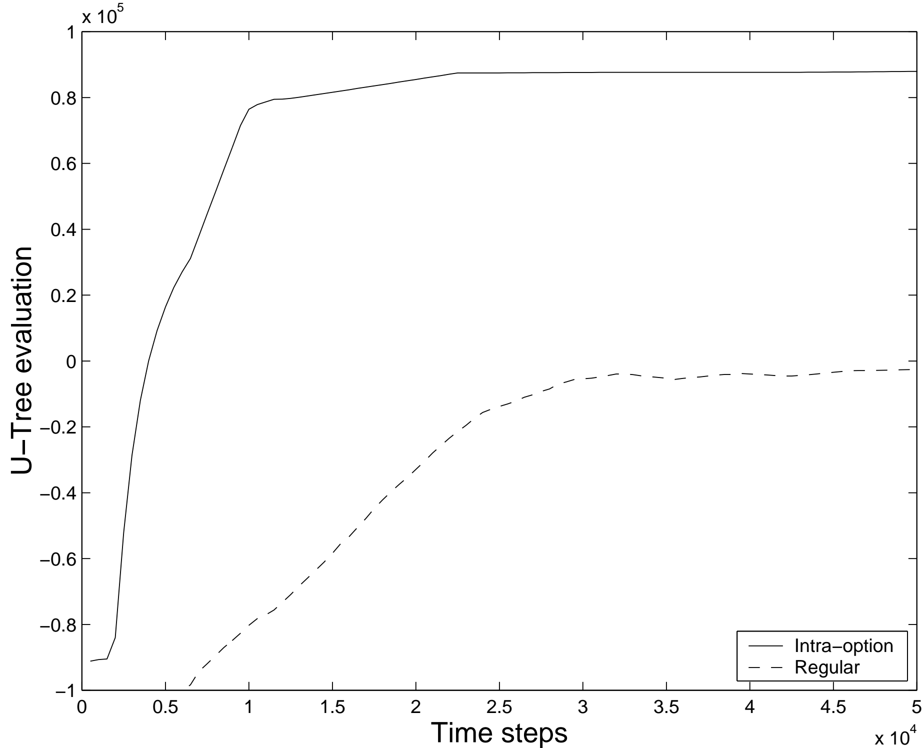

Figure 3.2. Comparison between intra-option and regular state abstraction

more quickly during intra-option state abstraction. Higher performance is achieved because the amount of evidence is larger. The target of an option is only reached once during each execution of the option, whereas it might be reached several times during the execution of another option.

In the second set of experiments, we performed 10 learning runs, each with a duration of 200,000 time steps. Figure 3.3 shows an example of the resulting U-Trees. Nodes that represent distinctions are drawn as circles, and leaf nodes are shown as squares or, in most cases, omitted. In the figure, a denotes a distinction over the previously executed option (in the order `Navigate`(q), `Pick-up` and `Drop-off`), and other letters denote a distinction over the corresponding observation. Note that the U-Tree of `Navigate(1)` did not make a distinction between x-positions in the lower part of the grid. In some places, for example in `Navigate(4)`, the right branch of x, the algorithm made a suboptimal distinction. A distinction over y would have given

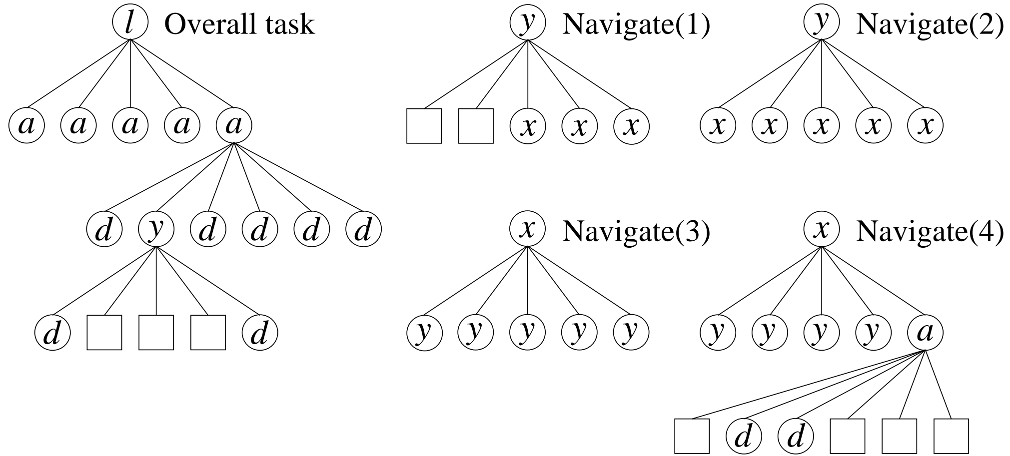

Figure 3.3. Learned U-Trees for different policies

a smaller number of leaves and would have been sufficient to represent an optimal policy. The U-Trees in the figure contain a total of 188 leaf nodes. Across 10 runs, the number of leaf nodes varied from 154 to 259, with an average of 189. Some leaf nodes were never visited, making the actual number of states even smaller. This is comparable to the results of Dietterich (2000a) who hand-coded a representation containing 106 states. Compared to the 500 distinct states of the Taxi task, or the 2,500 distinct states necessary to store the policies of the four navigation tasks and the overall task without state abstraction, our result is a significant improvement. Certainly, the memory required to store transition instances should also be taken into account. However, we believe that the memory savings due to option-specific state abstraction in larger tasks will significantly outweigh the memory requirement for U-Trees.

To show the benefits of option-specific state abstraction, we ran another experiment to compare the learning performance in the Taxi task of the U-Trees in Figure 3.3 with flat learning (i.e., without options) and hierarchical learning without state abstraction. We used SMDP Q-learning (Bradtke and Duff, 1995), which reduces to regular Q-learning in the flat case, to learn the policies of each option. We used an

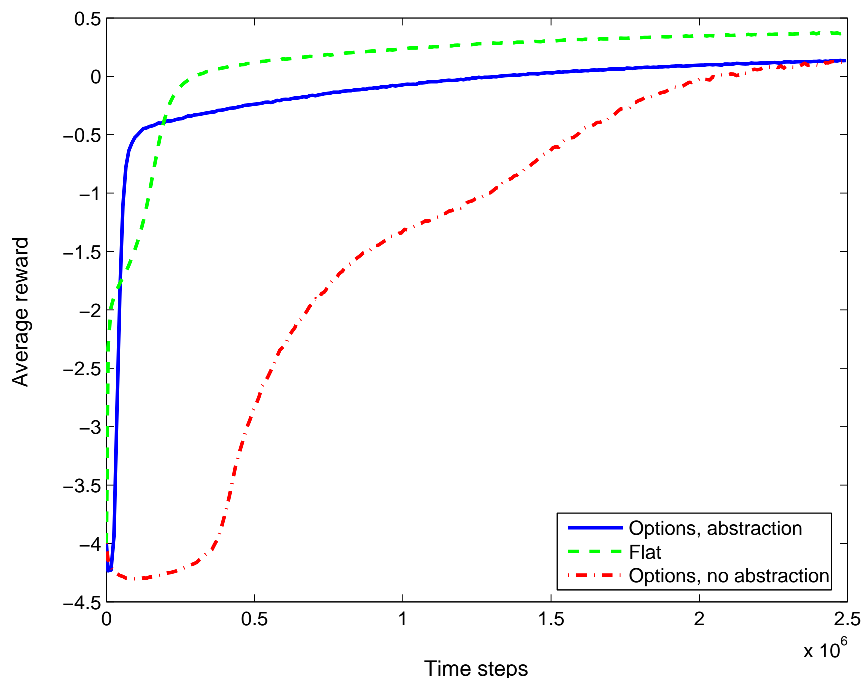

Figure 3.4. Comparison between hierarchical learning, with and without state abstraction, and flat learning

ϵ-greedy exploration scheme with $\epsilon = 0.1$, and decayed ϵ by 0.999 every 1,000 time steps. In the case of hierarchical learning without state abstraction, we used the same options as in the previous experiments. However, when there is no state abstraction, SMDP Q-learning has to estimate an action-value for each state-action pair instead of each leaf-action pair.

The results of the experiment appear in Figure 3.4. Each graph shows the average reward over 100 trials, plotted over the number of time steps. Hierarchical learning with option-specific state abstraction converges faster than the other approaches, taking into account that reward continues to increase slowly as ϵ is decayed. There are two possible explanations for why the reward level is slightly higher in the case of flat learning. The U-Trees in Figure 3.3 make several incorrect distinctions that may prevent reinforcement learning from finding an optimal policy. Also, the ϵ-greedy action selection impacts the reward more negatively in the case of hierarchical learning. Ex-

Table 3.1. Parameters in our implementation of the U-Tree algorithm

PARAMETER	FUNCTION
H	history index
K	interval with which refinements are considered
L	maximum length of history of transition instances
L_0	minimum history length to consider refinements
z	depth of fringe nodes
M	minimum number of transition instances at fringe nodes
θ	threshold for retaining a refinement
θ_0	threshold when no refinement has been made
ϵ	exploration factor
τ	temperature of softmax action selection

ecuting the wrong option to completion results in a larger setback in terms of reward than executing a random action for one time step. Note that hierarchical learning without state abstraction performs considerably worse than flat learning. This experiment illustrates our point that introducing activities only improves the learning complexity if the stand-alone task associated with each activity is significantly easier to solve than the overall task.

3.3 Discussion

We can think of several modifications that would help improve the U-Tree algorithm. Our implementation of the U-Tree algorithm uses many parameters, illustrated in Table 3.1. Each of these parameters has to be tuned for optimal performance. The U-Tree algorithm would be much easier to use if there were fewer parameters. For example, recall that our DBN learning algorithm evaluates refinements as soon as a minimum number of data instances map to a leaf. If we used this type of criterion in the U-Tree algorithm, we could replace the parameters K, L, L_0, and M with a single parameter.

Also, the U-Tree algorithm does not go back and reconsider refinements already made in the tree. This is a significant drawback since additional evidence may suggest that another refinement would result in a better representation. A future improvement of the U-Tree algorithm would be to include a mechanism that goes back in the tree and reconsiders refinements. Another potential extension of the U-Tree algorithm is to consider another type of refinement that detects symmetry among the values of a state variable. This could result in an even more compact representation, along the lines of the work by Ravindran (2004).

3.4 Related work

The idea of state abstraction in MDPs dates back to Smith (1971), who used a partitioned state space to store test quantities similar to values. Dean and Givan (1997) used stochastic bisimulation homogeneity as a framework to analyze state abstraction in MDPs. Their analysis permits the notion of a minimal MDP which is constructed by finding the coarsest homogeneous refinement of the state space of the originial MDP. Ravindran (2004) recently formulated an algebraic approach to abstraction in MDPs based on homomorphisms, an old concept from automata theory. He introduced the definition of a homomorphic image of an MDP, and showed that a homomorphic image preserves optimality. The homomorphic image of an MDP forms a partition of the set of state-action pairs of the MDP, which enables more general abstraction than the minimal MDP of Dean and Givan (1997).

Several researchers have developed algorithms that perform state abstraction from experience with the environment. Model minimization (Dean and Givan, 1997) successively refines an initial partition of the state space of an MDP until each block of the partition satisfies stochastic bisimulation homogeneity, approximating the minimal MDP. Ravindran (2004) extended model minimization to partitions of the set of

state-action pairs of an MDP. The resulting algorithm finds an approximation of the minimal homomorphic image of an MDP.

Other algorithms take a similar approach: they start with a coarse representation of the state space of an MDP and successively refine the representation to find an approximately minimal MDP. The G-learning algorithm (Chapman and Kaelbling, 1991) constructs a tree-based partition of the state space of an MDP starting with a single root node. The algorithm refines the tree when the value estimate of leaf nodes differ with sufficiently small significance. Boutilier and Dearden (1994) developed an algorithm that refines a partition based on immediately relevant discriminants. Discriminants with the greatest impact on value are deemed most relevant, but the algorithm also considers other criteria. The Parti-game algorithm (Moore and Atkeson, 1995) and variable resolution discretization (Munos and Moore, 1999) construct discrete partitions of continuous state spaces. The Parti-game algorithm refines the partition when it fails to accurately predict transition probabilities. Variable resolution discretization uses two global measures, influence and variance, to refine the partition. The U-Tree algorithm (McCallum, 1995), which the H-Tree algorithm extends, also falls within this category of algorithms, although it is applicable in more general tasks modeled as POMDPs.

Dietterich (2000b) was the first to formalize the idea of performing state abstraction separately for each activity. He outlines several conditions under which it is safe to perform state abstraction in a hierarchy of activities. Some of these conditions coincide with the definition of a homomorphic image (Ravindran, 2004). Andre and Russell (2002) defined safe state abstraction for Programmable Hierarchical Abstract Machines, an extension of the HAM model of activities. These approaches requires the system designer to handcraft state abstraction, whereas the H-Tree algorithm performs state abstraction from experience. Ravindran and Barto (2003) later introduced the idea of relativized options, which are defined without an absolute frame

of reference. The authors extended the model minimization technique to the SMDP framework to perform state abstraction separately for each option.

At the time of the H-Tree algorithm, hierarchical memory was simultaneously suggested by Hernandez-Gardiol and Mahadevan (2001). The authors developed an algorithm called Hierarchical Suffix Memory that combines memory with activities to propagate delayed reward across long decision sequences. Their algorithm uses Nearest Sequence Memory and Utile Suffix Memory (McCallum, 1995) to handle memory, represents activities using Hierarchical Abstract Machines (Parr and Russell, 1998), and uses SMDP Q-learning (Bradtke and Duff, 1995) to learn policies of activities. Hierarchical Suffix Memory uses experience to add memory to the state description of the stand-alone task associated with each activity. This makes it possible to perform state abstraction separately for each activity, just like the H-Tree algorithm. However, Nearest Sequence Memory and Utile Suffix Memory do not allow factored representations, so Hierarchical Suffix Memory is more likely to suffer from the curse of dimensionality in factored POMDPs.

Intra-option state abstraction is analogous to intra-option learning (Sutton et al., 1998). During intra-option learning, reinforcement learning updates occur as normal for the current option following execution of an action or option. If the same action or option would have had a non-zero probability of being selected by the policies of other options, reinforcement learning updates occur for those other options as well. Since experience accumulates faster, intra-option learning can significantly speed up the convergence time of learning.

CHAPTER 4

A CAUSAL APPROACH TO HIERARCHICAL DECOMPOSITION

One of the fundamental benefits of hierarchical decomposition is the ability to simplify the learning of a task by breaking it into suitable subtasks. However, indiscriminately introducing subtasks that are as difficult to solve as the overall task can significantly increase the complexity of learning. In the previous chapter we argued that to take full advantage of hierarchical decomposition, it is necessary to perform state abstraction separately for each activity. Our work used the option model of activity, and we showed how option-specific state abstraction in a hierarchy of options can reduce, instead of increase, the complexity of learning a task.

Our previous research on option-specific state abstraction assumes that a hierarchy of options is already given at the onset of learning. In other words, it ignores the problem of identifying suitable subtasks and relies on the system designer to decompose the task. However, it may not always be apparent to a system designer how to correctly identify appropriate subtasks prior to learning. Experience from interaction with the environment may provide additional hints about promising candidates for activities. We argue that an autonomous agent should be able to perform hierarchical decomposition on its own using experience from the environment.

In this chapter, we present Variable Influence Structure Analysis, or VISA, an algorithm that uses analysis of causal relationships between state variables to perform hierarchical decomposition of factored MDPs. The VISA algorithm uses the DBN model of factored MDPs to compactly represent transition probabilities and expected

reward. However, the VISA algorithm makes additional use of the DBN model. In addition to describing the effect of actions, the DBN model implicitly expresses a notion of causality between state variables, conditional on the actions. A specific value of one state variable can cause the value of another state variable to change if the right action is executed. The VISA algorithm uses information about this type of causal relationship to identify subtasks that change the values of state variables, and introduces an option for each subtask.

Even though the VISA algorithm identifies subtasks that enable hierarchical decomposition of factored MDPs, the true strength of the algorithm lies in its ability to perform option-specific state abstraction. The VISA algorithm constructs a causal graph that illustrates how the state variables of a factored MDP influence each other. Recall that each option is associated with an option SMDP that implicitly defines the option policy. Because of the causal relationships between state variables, the option SMDP of an option that causes the value of a particular state variable to change only needs to discriminate between state variables that influence that particular state variable. In addition, the option SMDP only needs to contain actions or options that cause the values of those influencing state variables to change. The resulting abstraction enables an important reduction in the complexity of learning the policies of the options identified by VISA.

Throughout this chapter, we assume that a DBN model of factored MDPs is provided prior to learning. Although this is an optimistic assumption, several other researchers have developed efficient algorithms for solving factored MDPs that also assume that a DBN model is given. It is not entirely unrealistic to assume that an autonomous agent is aware of the causes and effects associated with its actions prior to learning. Nevertheless, in Chapter 6 we relax the assumption and address the problem of learning a DBN model from experience with the environment.

4.1 The VISA algorithm

An autonomous agent can often anticipate how the variables describing the state of its environment are related as a result of executing actions. For example, a soccer-playing agent that is next to the ball and makes a motion to kick can influence the location of the ball. The variable describing the location of the ball and the variable describing the location of the agent are related as a result of kicking. The relationship between state variables can usually be described in terms of causes and effects. Kicking, while next to the ball, causes the location of the ball to change. There is a causal relationship between the variable describing the location of the agent and the variable describing the location of the ball, conditional on the action of kicking. In addition, variables are often conditionally independent. A soccer player's location on the field is independent of many variables, such as the dimensions of the field, the length of the grass, and the weather, regardless of which action is executed.

Since actions only sometimes cause the values of variables to change, it is useful to introduce activities that satisfy the conditions necessary for change. In the soccer example, a useful activity would be to take the agent next to the ball before kicking. The VISA algorithm searches for variable values and actions that cause the value of a state variable to change, and introduces an option for each such variable value change. The causal relationships between state variables and the conditional independence implicit in the lack of causal relationships offer ample opportunity to perform option-specific state abstraction, resulting in an important reduction in learning complexity. The details of the algorithm are described in the following sections.

4.1.1 Causal graph

The first step of the VISA algorithm is to construct a causal graph representing the causal relationships between state variables. The causal graph contains one node per state variable plus one node corresponding to expected reward. There is a directed

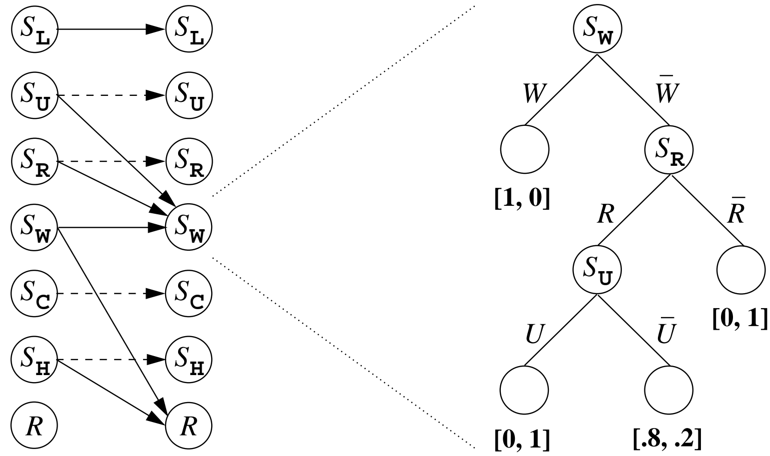

Figure 4.1. The DBN for action GO in the coffee task

edge between two state variables S_j and S_i if and only if there exists an action $a \in A$ such that there is an edge between S_j and S_i in the DBN for a. In other words, each edge in the causal graph represents a causal relationship between two state variables conditional on one or several actions. The algorithm labels each edge in the causal graph with the actions that give rise to the causal relationship.

We will use the coffee task (Boutilier et al., 1995) to illustrate the VISA algorithm. The coffee task is described by six binary state variables: $S_{\mathtt{L}}$, the robot's location (office or coffee shop); $S_{\mathtt{U}}$, whether the robot has an umbrella; $S_{\mathtt{R}}$, whether it is raining; $S_{\mathtt{W}}$, whether the robot is wet; $S_{\mathtt{C}}$, whether the robot has coffee; and $S_{\mathtt{H}}$, whether the user has coffee. The robot has four actions: `GO`, causing its location to change and the robot to get wet if it is raining and it does not have an umbrella; `BC` (buy coffee) causing it to hold coffee if it is in the coffee shop; `GU` (get umbrella) causing it to hold an umbrella if it is in the office; and `DC` (deliver coffee) causing the user to hold coffee if the robot has coffee and is in the office. All actions have a chance of failing. The robot gets a reward of 0.9 whenever the user has coffee plus a reward of 0.1 whenever it is dry.

Figure 4.1 shows the DBN for action GO in the coffee task. There are several interesting things to note. For each state variable S_i, there is an edge from the node representing the value of S_i prior to executing GO to the node representing the value of S_i after executing GO. In other words, each node in the causal graph should have an associated reflexive edge. However, we are not interested in the causal relationship of a state variable onto itself, so we remove reflexive edges in the causal graph. Also, there are edges from state variable $S_{\mathtt{U}}$ to state variable $S_{\mathtt{W}}$ in the DBN, as well as from $S_{\mathtt{R}}$ to $S_{\mathtt{W}}$. Consequently, there should be an edge from $S_{\mathtt{U}}$ to $S_{\mathtt{W}}$ in the causal graph labeled GO, as well as an edge from $S_{\mathtt{R}}$ to $S_{\mathtt{W}}$ labeled GO.

The causal graph of the coffee task is illustrated in Figure 4.2. Note that the edges from the DBN for action GO have been incorporated into the causal graph, as well as edges from the DBNs of the other actions. Also note that there are no cycles in the causal graph. However, this is not true in general for arbitrary tasks, since it is possible for state variables to mutually influence each other. The VISA algorithm gets rid of cycles in the causal graph by computing the strongly connected components of the causal graph. Each strongly connected component consists of one or several state variables that are pairwise connected through directed paths. It is possible to construct a component graph in which each node is a strongly connected component, and which has an edge between two nodes if and only if there is an edge in the causal graph between a state variable of the first component and a state variable of the second component. The component graph is guaranteed to contain no cycles. In the coffee task, each state variable in the causal graph is its own strongly connected component, so the component graph is identical to the causal graph.

4.1.2 Identifying exits

The VISA algorithm builds on ideas from the HEX-Q algorithm (Hengst, 2002), an algorithm that also performs hierarchical decomposition of factored MDPs from

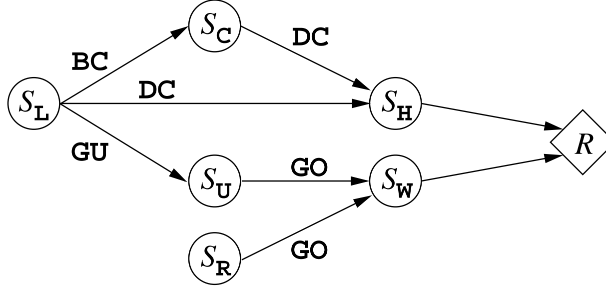

Figure 4.2. The causal graph of the coffee task

experience with the environment. The HEX-Q algorithm first determines an ordering on the state variables by randomly executing actions and counting the frequency with which the value of each state variable changes. The state variable whose value changes the most frequently becomes the lowest variable in the ordering, and so on. For each state variable S_i in the ordering, the HEX-Q algorithm identifies exits $\langle k, a \rangle$, pairs of a state variable value $k \in Val(S_i)$ and an action $a \in A$, that cause the value of the next state variable in the ordering to change. The HEX-Q algorithm introduces an option for each exit state, and the options on one level of the hierarchy become actions on the next level.

Even though the HEX-Q algorithm achieved some early success, the frequency of change heuristic may not be an accurate indicator of how state variables influence each other. In addition, the ordering does not capture the fact that the value of a state variable may depend on multiple other state variables. Figure 4.3 illustrates the state variable ordering that the HEX-Q algorithm comes up with in the coffee task. There are several differences between this ordering and the SVIG. The ordering wrongly concludes that state variable $S_{\mathtt{W}}$ influences $S_{\mathtt{R}}$, when it is really the other way around. The ordering also fails to capture the fact that the value of $S_{\mathtt{H}}$ depends on both $S_{\mathtt{L}}$ and $S_{\mathtt{C}}$.

The VISA algorithm also searches for exits that cause the values of state variables to change. However, instead of the frequency of change heuristic, the VISA algorithm

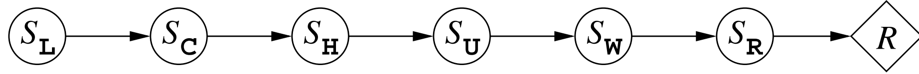

Figure 4.3. HEX-Q's state variable ordering in the coffee task

uses the causal graph to determine how state variables influence each other. Since the causal graph more realistically describes the causal relationships between state variables, the VISA algorithm is able to successfully decompose more general tasks than the HEX-Q algorithm. Also, since the value of a state variable may depend on several other state variables, an exit $\langle \mathbf{c}, a \rangle$ in the VISA algorithm is composed of a context $\mathbf{c}$ and an action $a \in A$. Recall that a context $\mathbf{c}$ is an assignment of values to a subset $\mathbf{C} \subseteq \mathbf{S}$ of the state variables.

The VISA algorithm searches for exits in the conditional probability trees of the DBN model. Consider, for example, the conditional probability tree associated with state variable $S_{\mathtt{W}}$ and action `GO` in Figure 4.1. The left-most leaf of the tree is associated with states that assign the value W to state variable $S_{\mathtt{W}}$. As a result of executing action `GO` in such states, the value of $S_{\mathtt{W}}$ becomes W with probability 1. In other words, if the robot is wet prior to executing `GO`, it will always remain wet, so the value of $S_{\mathtt{W}}$ does not change. The right-most leaf of the tree is associated with states that assign $\overline{W}$ to $S_{\mathtt{W}}$ and $\overline{R}$ to $S_{\mathtt{R}}$. As a result of executing action `GO` in such states, the value of $S_{\mathtt{W}}$ becomes $\overline{W}$ with probability 1. In other words, if the robot is dry prior to executing `GO` and it is not raining, it will always remain dry, so again the value of $S_{\mathtt{W}}$ does not change. The VISA algorithm does not generate exits for either of these two leaves.

Now consider the leaf associated with states that assign $\overline{W}$ to $S_{\mathtt{W}}$, R to $S_{\mathtt{R}}$, and $\overline{U}$ to $S_{\mathtt{U}}$. As a result of executing action `GO` in such states, the value of $S_{\mathtt{W}}$ becomes W with probability 0.8 and $\overline{W}$ with probability 0.2. Since the value of state variable $S_{\mathtt{W}}$ changes from $\overline{W}$ to W with non-zero probability, the VISA algorithm generates

Table 4.1. Exits identified in the coffee task

EXIT	VARIABLE	CHANGE
$\langle(), \texttt{GO}\rangle$	$S_{\texttt{L}}$	$\overline{L} \rightarrow L,\ L \rightarrow \overline{L}$
$\langle(S_{\texttt{L}} = L), \texttt{BC}\rangle$	$S_{\texttt{C}}$	$\overline{C} \rightarrow C$
$\langle(S_{\texttt{L}} = \overline{L}), \texttt{DC}\rangle$	$S_{\texttt{C}}$	$C \rightarrow \overline{C}$
$\langle(S_{\texttt{L}} = \overline{L}, S_{\texttt{C}} = C), \texttt{DC}\rangle$	$S_{\texttt{H}}$	$\overline{H} \rightarrow H$
$\langle(S_{\texttt{L}} = \overline{L}), \texttt{GU}\rangle$	$S_{\texttt{U}}$	$\overline{U} \rightarrow U$
$\langle(S_{\texttt{U}} = \overline{U}, S_{\texttt{R}} = R), \texttt{GO}\rangle$	$S_{\texttt{W}}$	$\overline{W} \rightarrow W$

an exit $\langle(S_{\texttt{U}} = \overline{U}, S_{\texttt{R}} = R), \texttt{GO}\rangle$ that causes the value of $S_{\texttt{W}}$ to change. Note that the value of $S_{\texttt{W}}$ does not appear in the exit since that is the state variable whose value we are trying to change. Also note that the exit $\langle(S_{\texttt{U}} = \overline{U}, S_{\texttt{R}} = R), \texttt{GO}\rangle$ does not cause the value of $S_{\texttt{W}}$ to change with probability 1, so to effectuate the change the robot may have to execute action GO multiple times in the context $(S_{\texttt{U}} = \overline{U}, S_{\texttt{R}} = R)$.

Table 4.1 shows a complete list of exits identified by the VISA algorithm in the coffee task. The table shows which state variable is affected by each exit together with the resulting change. To generate these exits the VISA algorithm had to search through each leaf of each conditional probability tree of the DBN model. At each leaf, the algorithm examines whether the value of state variable S_i changes, where S_i is the state variable whose conditional probabilities the current tree represents. In other words, the complexity of this part of the algorithm is proportional to the number of leaves of the conditional probability trees.

4.1.3 Introducing options

For each exit $\langle \mathbf{c}, a\rangle$ with a non-empty context $\mathbf{c}$, the VISA algorithm introduces an option $o = \langle I, \pi, \beta\rangle$. Option o terminates in any state $\mathbf{s} \in S$ whose projection $f_{\mathbf{C}}(\mathbf{s})$ onto $\mathbf{C}$ equals $\mathbf{c}$. We refer to the options introduced by the VISA algorithm as exit options. Unlike regular options, an exit option associated with an exit $\langle \mathbf{c}, a\rangle$ executes action a following termination. Note that it is not necessary to introduce options

for exits with empty contexts, since these options are in fact equivalent to primitive actions. For example, the VISA algorithm identifies an exit $\langle (), \texttt{GO} \rangle$ in the coffee task. Executing action GO in any state causes the location of the robot to change, so the option associated with this exit is equivalent to the primitive action GO. As we shall see, it is still useful to detect exits with empty contexts.

In the coffee task example, we will adopt the convention of refering to an exit option using the change that it causes, since this is an unambiguous and simple notation. For example, option $\overline{W} \rightarrow W$ is the exit option associated with the exit $\langle (S_{\texttt{U}} = \overline{U}, S_{\texttt{R}} = R), \texttt{GO} \rangle$ that causes the value of $S_{\mathbf{W}}$ to change from $\overline{W}$ to W with non-zero probability. In general, several exits may cause the same change in the value of a variable, and the VISA algorithm would introduce an exit option for each of these exits, so this notation would no longer be unambiguous.

4.1.4 Initiation set

Two factors influence the initiation set I of an exit option o. Option o should only be admissible in states from which it is possible to reach the associated context $\mathbf{c}$. For example, option $\overline{W} \rightarrow W$ should only be admissible in states that assign $\overline{U}$ to $\mathbf{S}_{\texttt{U}}$ and R to $\mathbf{S}_{\texttt{R}}$. The robot has no action for getting rid of an umbrella, and it cannot affect whether it is raining, so it can only get wet if it does not have an umbrella and it is raining. Option o should also only be admissible if its associated exit $\langle \mathbf{c}, a \rangle$ causes the value of at least one state variable to change. In our example, option $\overline{W} \rightarrow W$ should only be admissible in states that assign $\overline{W}$ to $\mathbf{S}_{\texttt{W}}$, since otherwise the option cannot cause the value of $\mathbf{S}_{\texttt{W}}$ to change from $\overline{W}$ to W.

The VISA algorithm uses a sophisticated method to construct the initiation set I of an exit option o. For each strongly connected component, the algorithm constructs a transition graph that represents possible transitions between contexts in the joint value set of its state variables. Each transition graph is in the form of a tree in which

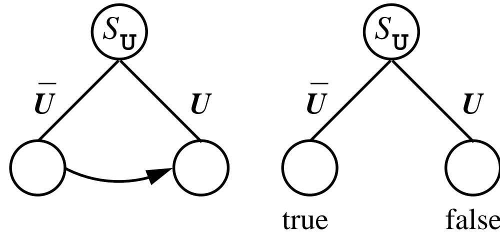

Figure 4.4. The transition graph (left) and reachability tree (right) of the strongly connected component containing $S_{\mathtt{U}}$

possible transitions are represented as directed edges between the leaves. Possible transitions are determined using the conditional probability trees of the DBN model. Figure 4.4 illustrates the transition graph of the strongly connected component containing the state variable $S_{\mathtt{U}}$ in the coffee task. The robot can acquire an umbrella by executing the exit option $\overline{U} \rightarrow U$, so there is a corresponding edge in the transition graph between the leaf associated with states that assign $\overline{U}$ to $S_{\mathtt{U}}$ and the leaf associated with states that assign U to $S_{\mathtt{U}}$. However, the robot has no action for getting rid of an umbrella, so there is no edge going the other way.

The VISA algorithm uses the transition graphs to construct a tree that classifies states on the basis of whether or not the context $\mathbf{c}$ of the exit $\langle \mathbf{c}, a \rangle$ associated with option o is reachable. The algorithm starts at leaves corresponding to the context $\mathbf{c}$ and uses depth-first search, traversing the edges backwards, to determine reachability. In addition to the transition graph, Figure 4.4 also illustrates the reachability tree associated with the strongly connected component containing $S_{\mathtt{U}}$. The reachability tree determines whether (true) or not (false) the context $(S_{\mathtt{U}} = \overline{U}, S_{\mathtt{R}} = R)$ of the exit $\langle (S_{\mathtt{U}} = \overline{U}, S_{\mathtt{R}} = R), \mathtt{GO} \rangle$ associated with option $\overline{W} \rightarrow W$ is reachable from different states. If the context $\mathbf{c}$ contains values of state variables from different strongly connected components, the VISA algorithm constructs a reachability tree for each of the components. In our example, the VISA algorithm constructs another reachability

tree associated with the strongly connected component containing the state variable $S_{\mathtt{R}}$.

In a similar way, the VISA algorithm constructs a tree that classifies states on the basis of whether or not the associated exit changes the value of at least one state variable in the corresponding strongly connected component. This tree can also be constructed using the conditional probability trees of the DBN model. In our example, states that assign $\overline{W}$ to $S_{\mathtt{W}}$ map to a leaf labeled true, and states that assign W to $S_{\mathtt{W}}$ map to a leaf labeled false, since the exit $\langle(S_{\mathtt{U}} = \overline{U}, S_{\mathtt{R}} = R), \mathtt{GO}\rangle$ does not cause the value of $S_{\mathtt{W}}$ to change if its current value is W. The initiation set I of option o is implicitly defined by the trees constructed by VISA. A state $\mathbf{s} \in S$ is an element in I if and only if $\mathbf{s}$ maps to a leaf labeled true in each tree.

4.1.5 Termination condition

An exit option o terminates as soon as it reaches the context $\mathbf{c}$ of its associated exit $\langle \mathbf{c}, a\rangle$, or as soon as it can no longer reach $\mathbf{c}$. Even though an exit option executes action a following termination, we can still represent termination of the option using the standard termination condition function β. For an exit option o, $\beta(\mathbf{s})$ is 1 for states in the set $\{\mathbf{s} \in S \mid f_{\mathbf{C}}(\mathbf{s}) = \mathbf{c}\}$, where $\mathbf{c}$ is the associated context. $\beta(\mathbf{s})$ is also 1 for states $\mathbf{s} \notin I$, i.e., when the process can no longer reach the associated context $\mathbf{c}$. In all other cases, $\beta(\mathbf{s}) = \mathbf{0}$.

4.1.6 Policy

The VISA algorithm cannot directly define the policy π of exit option o since it does not know the best strategy for reaching the associated context $\mathbf{c}$. Instead, the algorithm constructs an option SMDP $\mathcal{M}_o = \langle S_o, O_o, \Psi_o, P_o, R_o\rangle$ for option o that implicitly defines its policy π. First of all, the algorithm defines $S_o = S$. Next, the algorithm finds all strongly connected components that contain at least one state variable whose value appears in the context $\mathbf{c}$ associated with option o. The algorithm

defines O_o as the set of options that cause the values of state variables in those strongly connected components to change. For example, the exit option $\overline{W} \rightarrow W$ is associated with the context $(S_{\texttt{U}} = \overline{U}, S_{\texttt{R}} = R)$. Two strongly connected components contain state variables whose values appear in the context: the strongly connected component containing $S_{\texttt{U}}$, and the strongly connected component containing $S_{\texttt{R}}$. A single option, $\overline{U} \rightarrow U$, causes the values of state variables to change in the former component, while no option causes the values of state variables to change in the latter. In other words, the option set O_o of $\overline{W} \rightarrow W$ only needs to include the exit option $\overline{U} \rightarrow U$. Note that primitive actions may affect the values of state variables in strongly connected components for which there are no options; for example, action GO affects the value of state variable $S_{\texttt{L}}$.

If there are lower-level options that cause the process to leave the initiation set of an option in O_o, the VISA algorithm includes these options in O_o as well. For example, the exit option $\overline{U} \rightarrow U$ causes the process to leave the initiation set of the exit option $\overline{W} \rightarrow W$. If the robot does not have an umbrella and it is raining, the exit option $\overline{W} \rightarrow W$ will no longer be admissible as a result of executing the exit option $\overline{U} \rightarrow U$ causing the robot to hold an umbrella. In other words, an option whose option set O_o includes the exit option $\overline{W} \rightarrow W$ should include the exit option $\overline{U} \rightarrow U$ as well.

The VISA algorithm defines the expected reward function R_o as -1 everywhere except when option o terminates unsuccessfully, in which case the algorithm administers a large negative reward. This ensures that the policy π of option o attempts to reach the context $\mathbf{c}$ as quickly as possible. The set of admissible state-option pairs, Ψ_o, is determined by the initiation sets of the options in O_o. The VISA algorithm does not represent the transition probability function P_o explicitly. It is possible to construct a DBN model for each option similar to the DBN model for the primitive actions. However, there is currently no technique that constructs DBN models of

options without enumerating all states. Since the whole point of the VISA algorithm is to alleviate the curse of dimensionality, we want to avoid enumerating the states. Instead, the VISA algorithm uses reinforcement learning, which does not require explicit knowledge of the transition probabilities, to learn the policy π of option o. In Chapter 5, we develop an algorithm that constructs DBN models of options identified by the VISA algorithm without enumerating all states. The transition graphs of strongly connected components play a part in constructing DBN models of options, but nothing prevents options from changing the values of other state variables as well.

4.1.7 State abstraction

The VISA algorithm simplifies learning the option SMDPs by performing state abstraction separately for each exit option. This is where causality really matters. Let us consider all strongly connected components that contain at least one state variable whose value appears in the context $\mathbf{c}$ associated with option o. Let $\mathbf{Z} \subseteq \mathbf{S}$ denote the subset of state variables contained in those strongly connected components. Let $\mathbf{Y} \subseteq \mathbf{S}$ denote the subset of state variables S_i such that either $S_i \in \mathbf{Z}$ or such that there is a directed path in the causal graph from S_i to a state variable in $\mathbf{Z}$. For example, in case of exit option $\overline{W} \rightarrow W$, $\mathbf{Z} = \{S_{\mathtt{U}}, S_{\mathtt{R}}\}$ and $\mathbf{Y} = \{S_{\mathtt{L}}, S_{\mathtt{U}}, S_{\mathtt{R}}\}$, since there is a directed path from $S_{\mathtt{L}}$ to $S_{\mathtt{U}}$ in the causal graph of the coffee task.

Recall that the goal of exit option o is to reach the context $\mathbf{c}$. We know that $\mathbf{C} \subseteq \mathbf{Z} \subseteq \mathbf{Y}$, i.e., that the state variables whose values appear in the context $\mathbf{c}$ associated with option o are contained in $\mathbf{Y}$. We also know that there are no edges from any state variable $S_j \notin \mathbf{Y}$ to any state variable $S_i \in \mathbf{Y}$; if there were, state variable S_j would have been included in $\mathbf{Y}$. It follows that the option SMDP $\mathcal{M}_o$ can ignore the values of state variables not in $\mathbf{Y}$, since they have no influence on the variables in $\mathbf{C}$ whose values we want to set to $\mathbf{c}$.

More formally, we can define a partition that satisfies the stochastic substitution property and is reward respecting, and thus guaranteed to preserve an optimal solution to the option SMDP $\mathcal{M}_o$.

Theorem 4.1.1 *The projection function $f_{\mathbf{Y}}$ induces a partition $\Lambda_{\mathbf{Y}}$ of S that has the stochastic substitution property and is reward respecting.*

Proof The projection function $f_{\mathbf{Y}}$ induces a partition $\Lambda_{\mathbf{Y}}$ of S such that two states $\mathbf{s}_1$ and $\mathbf{s}_2$ belong to the same block if and only if $f_{\mathbf{Y}}(\mathbf{s}_1) = f_{\mathbf{Y}}(\mathbf{s}_2)$, i.e., if $\mathbf{s}_1$ and $\mathbf{s}_2$ assign exactly the same values to state variables in $\mathbf{Y}$. Let $\mathbf{y}_\lambda$ denote the assignment to $\mathbf{Y}$ of states in block λ of the induced partition, i.e., for each state $\mathbf{s} \in \lambda$, we have that $f_{\mathbf{Y}}(\mathbf{s}) = \mathbf{y}_\lambda$. Then for each pair of states $(\mathbf{s}_1, \mathbf{s}_2) \in S^2$, each action $a \in A$, and each block $\lambda \in \Lambda_{\mathbf{Y}}$, $[\mathbf{s}_1]_{\Lambda_{\mathbf{Y}}} = [\mathbf{s}_2]_{\Lambda_{\mathbf{Y}}}$ implies that

$$\begin{aligned}
\sum_{\mathbf{s}\in\lambda} P(\mathbf{s} \mid \mathbf{s}_1, a) &= \sum_{\mathbf{s}\in\lambda} P(f_{\mathbf{Y}}(\mathbf{s}) \mid \mathbf{s}_1, a) P(f_{\mathbf{S}-\mathbf{Y}}(\mathbf{s}) \mid \mathbf{s}_1, a) = \\
&= \sum_{\mathbf{s}\in\lambda} P(\mathbf{y}_\lambda \mid f_{\mathbf{Y}}(\mathbf{s}_1), a) P(f_{\mathbf{S}-\mathbf{Y}}(\mathbf{s}) \mid \mathbf{s}_1, a) = \\
&= P(\mathbf{y}_\lambda \mid f_{\mathbf{Y}}(\mathbf{s}_1), a) \sum_{\mathbf{s}\in\lambda} P(f_{\mathbf{S}-\mathbf{Y}}(\mathbf{s}) \mid \mathbf{s}_1, a) = \\
&= P(\mathbf{y}_\lambda \mid f_{\mathbf{Y}}(\mathbf{s}_2), a) \sum_{\mathbf{s}\in\lambda} P(f_{\mathbf{S}-\mathbf{Y}}(\mathbf{s}) \mid \mathbf{s}_2, a) = \\
&= \sum_{\mathbf{s}\in\lambda} P(\mathbf{y}_\lambda \mid f_{\mathbf{Y}}(\mathbf{s}_2), a) P(f_{\mathbf{S}-\mathbf{Y}}(\mathbf{s}) \mid \mathbf{s}_2, a) = \\
&= \sum_{\mathbf{s}\in\lambda} P(f_{\mathbf{Y}}(\mathbf{s}) \mid \mathbf{s}_2, a) P(f_{\mathbf{S}-\mathbf{Y}}(\mathbf{s}) \mid \mathbf{s}_2, a) = \sum_{\mathbf{s}\in\lambda} P(\mathbf{s} \mid \mathbf{s}_2, a).
\end{aligned}$$

The equality $\sum_{\mathbf{s}\in\lambda} P(f_{\mathbf{S}-\mathbf{Y}}(\mathbf{s}) \mid \mathbf{s}_1, a) = \sum_{\mathbf{s}\in\lambda} P(f_{\mathbf{S}-\mathbf{Y}}(\mathbf{s}) \mid \mathbf{s}_2, a)$ follows from the fact that as we sum over states in λ, we go through every possible assignment of values to state variables in the set $\mathbf{S} - \mathbf{Y}$, so in fact, $\sum_{\mathbf{s}\in\lambda} P(f_{\mathbf{S}-\mathbf{Y}}(\mathbf{s}) \mid \mathbf{s}', a) = 1$ for each state $\mathbf{s}' \in S$. It follows that the partition $\Lambda_{\mathbf{Y}}$ induced by $f_{\mathbf{Y}}$ has the stochastic substitution property.

In general, the partition $\Lambda_{\mathbf{Y}}$ induced by $f_{\mathbf{Y}}$ is not reward respecting with respect to the expected reward function R of the original MDP. However, recall that the expected reward function R_o of option o is independent of the expected reward function R of the original MDP. To form a reduced option SMDP it is sufficient that the partition $\Lambda_{\mathbf{Y}}$ is reward respecting with respect to R_o. R_o is defined as -1 everywhere except when the process leaves the initiation set of option o. The initiation set of option o is completely determined by the state variables in $\mathbf{Z} \subseteq \mathbf{Y}$, so whether or not the process leaves the initiation set depends only on those state variables. It follows that $\Lambda_{\mathbf{Y}}$ is reward respecting with respect to R_o.

The VISA algorithm goes a step further and forms the partition $\Lambda_{\mathbf{Z}}$ induced by the projection $f_{\mathbf{Z}}$. In other words, the option SMDP of option o ignores all state variables not in strongly connected components for which the value of at least one state variable appears in the context $\mathbf{c}$ associated with option o.

Theorem 4.1.2 *The projection function $f_{\mathbf{Z}}$ induces a partition $\Lambda_{\mathbf{Z}}$ of S that is reward respecting and has the stochastic substitution property if and only if for each pair of states $\mathbf{s}_1, \mathbf{s}_2 \in S^2$, each option $o' \in O_o$, and each block $\lambda \in \Lambda_{\mathbf{Z}}$, $[\mathbf{s}_1]_{\Lambda_{\mathbf{Z}}} = [\mathbf{s}_2]_{\Lambda_{\mathbf{Z}}}$ implies that $P_o(\mathbf{z}_\lambda \mid f_{\mathbf{Y}}(\mathbf{s}_1), o') = P_o(\mathbf{z}_\lambda \mid f_{\mathbf{Y}}(\mathbf{s}_2), o')$, where $\mathbf{z}_\lambda$ is the assignment of values to $\mathbf{Z}$ of states in block λ.*

Proof $\Lambda_{\mathbf{Z}}$ is still reward respecting with respect to R_o. However, a state variable in $\mathbf{Y} - \mathbf{Z}$ may influence the state variables in $\mathbf{Z}$, so $\Lambda_{\mathbf{Z}}$ does not always have the stochastic substitution property. We can write the sum $\sum_{\mathbf{s} \in \lambda} P_o(\mathbf{s} \mid \mathbf{s}_1, o')$ as

$$\begin{aligned}\sum_{\mathbf{s}\in\lambda} P_o(\mathbf{s} \mid \mathbf{s}_1, o') &= \sum_{\mathbf{s}\in\lambda} P_o(f_{\mathbf{Z}}(\mathbf{s}) \mid \mathbf{s}_1, o') P_o(f_{\mathbf{S}-\mathbf{Z}}(\mathbf{s}) \mid \mathbf{s}_1, o') = \\ &= \sum_{\mathbf{s}\in\lambda} P_o(\mathbf{z}_\lambda \mid f_{\mathbf{Y}}(\mathbf{s}_1), o') P_o(f_{\mathbf{S}-\mathbf{Z}}(\mathbf{s}) \mid \mathbf{s}_1, o') = \\ &= P_o(\mathbf{z}_\lambda \mid f_{\mathbf{Y}}(\mathbf{s}_1), o') \sum_{\mathbf{s}\in\lambda} P_o(f_{\mathbf{S}-\mathbf{Z}}(\mathbf{s}) \mid \mathbf{s}_1, o') = P_o(\mathbf{z}_\lambda \mid f_{\mathbf{Y}}(\mathbf{s}_1), o').\end{aligned}$$

Using the same calculations, we can obtain $\sum_{\mathbf{s}\in\lambda} P_o(\mathbf{s} \mid \mathbf{s}_2, o') = P_o(\mathbf{z}_\lambda \mid f_{\mathbf{Y}}(\mathbf{s}_2), o')$. $\Lambda_{\mathbf{Z}}$ has the stochastic substitution property if and only if for each pair of states $(\mathbf{s}_1, \mathbf{s}_2) \in S^2$, each option $o' \in O_o$, and each block $\lambda \in \Lambda_{\mathbf{Z}}$, $[\mathbf{s}_1]_{\Lambda_{\mathbf{Z}}} = [\mathbf{s}_2]_{\Lambda_{\mathbf{Z}}}$ implies that $P_o(\mathbf{z}_\lambda \mid f_{\mathbf{Y}}(\mathbf{s}_1), o') = P_o(\mathbf{z}_\lambda \mid f_{\mathbf{Y}}(\mathbf{s}_2), o')$, where $f_{\mathbf{Y}}(\mathbf{s}_1) = f_{\mathbf{Y}}(\mathbf{s}_2)$ does not hold in general.

Because of the work of Dean and Givan (1997) and Ravindran (2004), it follows from Theorem 4.1.1 that the partition $\Lambda_{\mathbf{Y}}$ induces a reduced SMDP that preserves optimality. Since the reduced SMDP has far fewer state-action pairs than the original option SMDP, it is significantly easier to solve, resulting in an important reduction in complexity. In addition, it follows from Theorem 4.1.2 that the partition $\Lambda_{\mathbf{Z}}$ induces a reduced SMDP that preserves optimality if and only if for each option $o' \in O_o$, state variables in $\mathbf{Y} - \mathbf{Z}$ do not influence the state variables in $\mathbf{Z}$ as a result of executing o'. Instead of solving the option SMDP directly, the VISA algorithm solves the reduced SMDP induced by the partition $\Lambda_{\mathbf{Z}}$, which has even fewer state-action pairs than the reduced SMDP induced by $\Lambda_{\mathbf{Y}}$.

Because of the way exits are defined, the exit options discovered by the VISA algorithm often satisfy the conditions necessary for the partition $\Lambda_{\mathbf{Z}}$ to have the stochastic substitution property. For example, consider the exit option $\overline{W} \rightarrow W$ in the coffee task. The only option in the set O_o is $\overline{U} \rightarrow U$, which is associated with the exit $\langle (S_{\mathtt{L}} = \overline{L}), \mathtt{GU} \rangle$. Recall that with respect to option $\overline{W} \rightarrow W$, $\mathbf{Z} = \{S_{\mathtt{U}}, S_{\mathtt{R}}\}$ and $\mathbf{Y} = \{S_{\mathtt{L}}, S_{\mathtt{U}}, S_{\mathtt{R}}\}$, so $\mathbf{Y} - \mathbf{Z} = \{S_{\mathtt{L}}\}$. As a result of executing action GU, the resulting value of state variable $S_{\mathtt{U}}$ depends on the previous value of state variable $S_{\mathtt{L}}$. However, as a result of executing the exit option $\overline{U} \rightarrow U$, the resulting value of $S_{\mathtt{U}}$ does not depend on the previous value of $S_{\mathtt{L}}$. Regardless of the previous value of $S_{\mathtt{L}}$, option $\overline{U} \rightarrow U$ always reaches the context $(S_{\mathtt{L}} = \overline{L})$ prior to executing GU, which causes the robot to pick up an umbrella with non-zero probability. It follows that the partition $\Lambda_{\mathbf{Z}}$ induced by $f_{\mathbf{Z}}$ has the stochastic substitution property.

If there exists a state variable in $\mathbf{Y} - \mathbf{Z}$ that influences a state variable in $\mathbf{Z}$, the partition $\Lambda_{\mathbf{Z}}$ does not have the stochastic substitution property. In other words, an optimal solution to the option SMDP $\mathcal{M}_o$ is not preserved in the reduced SMDP induced by the partition $\Lambda_{\mathbf{Z}}$. A solution to the reduced SMDP only corresponds to an approximate solution to $\mathcal{M}_o$. However, we believe that there is still a reason to perform state abstraction this way. The size of the partition $\Lambda_{\mathbf{Y}}$ may be exponentially larger than the size of $\Lambda_{\mathbf{Z}}$, so the difference in learning complexity may be significant in the two cases. We argue that the reduction in learning complexity often outweighs the loss of exact optimality.

To take even further advantage of structure, the VISA algorithm stores the policies of options in the form of policy trees. The benefit of using a policy tree is that the number of leaves in the tree may be smaller than the actual number of states. At each leaf of the policy tree, the VISA algorithm stores action-values or, more accurately described, option-values, which indicate the utility of executing different options in states that map to that leaf. Recall that the VISA algorithm maintains a transition graph, in the form of a tree, for each strongly connected component in the causal graph. The policy tree of an option can be constructed by merging the transition graph trees of strongly connected components that contain state variables whose values appear in the associated context. The policy tree induces a partition Λ_π such that $\Lambda_{\mathbf{Z}} \leq \Lambda_\pi$, i.e., the partition $\Lambda_{\mathbf{Z}}$ refines Λ_π.

Another part of abstraction is reducing the number of options in the option set O_o of the option SMDP $\mathcal{M}_o$. If there are fewer options to select from, an autonomous agent can discover more quickly which option or options result in an optimal value for each block of the state partition. As we explained above, the VISA algorithm finds strongly connected components that contain at least one state variable whose value appears in the context $\mathbf{c}$ associated with option o. The algorithm fills the option set O_o with options that change the values of state variables in those strongly connected

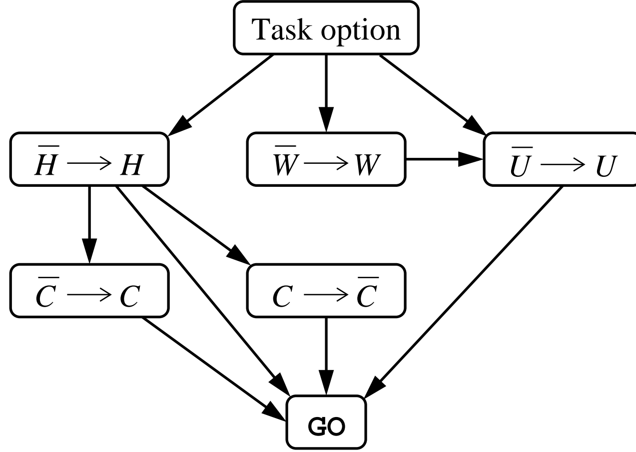

Figure 4.5. The hierarchy of options discovered by the VISA algorithm in the coffee task

components. The algorithm also includes options that leave the initiation sets of options in O_o. It is not necessary to include other options in O_o since they do not have any impact on reaching the context $\mathbf{c}$ associated with option o. Thus, the VISA algorithm limits the number of options of each option SMDP, further reducing the complexity of learning.

4.1.8 Task option

The VISA algorithm also introduces an option, which we call the task option, associated with the reward node in the component graph of the task. The algorithm uses the same strategy to construct the task option as the other options. However, the expected reward function of the task option SMDP is the same as the expected reward function of the original MDP. In addition, the task option SMDP only discriminates between state variables in strongly connected components that have edges to the reward node in the component graph. The learned policy of the task option is an approximate solution to the original MDP, which uses the other options discovered by the VISA algorithm. Figure 4.5 shows the hierarchy of options that the VISA algorithm comes up with in the coffee task.

4.1.9 Exit transformations

Sometimes it is possible to transform exits in order to take further advantage of causality. Consider the two exits $\langle(S_{\mathtt{L}} = \overline{L}), \mathtt{DC}\rangle$ and $\langle(S_{\mathtt{L}} = \overline{L}, S_{\mathtt{C}} = C), \mathtt{DC}\rangle$ in the coffee task. These exits are almost identical: their associated exit options both terminate in states that assign the value $\overline{L}$ to state variable $S_{\mathtt{L}}$ and execute action DC following successful termination. Recall that $C \to \overline{C}$ is the exit option associated with the exit $\langle(S_{\mathtt{L}} = \overline{L}), \mathtt{DC}\rangle$, causing the value of $S_{\mathtt{C}}$ to change from C to $\overline{C}$. The effect of the exit $\langle(S_{\mathtt{L}} = \overline{L}, S_{\mathtt{C}} = C), \mathtt{DC}\rangle$ is equivalent to the effect of a transformed exit $\langle(S_{\mathtt{C}} = C), C \to \overline{C}\rangle$, i.e., reach a state that assigns C to $S_{\mathtt{C}}$ and execute option $C \to \overline{C}$ following termination. The benefit of this transformation is that the exit option $\overline{H} \to H$ associated with the exit $\langle(S_{\mathtt{L}} = \overline{L}, S_{\mathtt{C}} = C), \mathtt{DC}\rangle$ no longer has to care about the value of $S_{\mathtt{L}}$, effectively removing an edge in the component graph of the task.

After identifying an exit, the VISA algorithm compares it to exits identified for previous strongly connected components. If an exit transformation is possible, the VISA algorithm performs the transformation and updates the set $\mathbf{Z}$ of state variables in strongly connected components whose state variables appear in the context of the exit. Recall that the VISA algorithm performs state abstraction by constructing the partition $\Lambda_{\mathbf{Z}}$ induced by the projection $f_{\mathbf{Z}}$ onto the joint value set of the state variables in $\mathbf{Z}$. Exit transformations reduce the number of state variables in $\mathbf{Z}$, thus reducing the complexity of solving the associated option SMDP.

4.1.10 Merging strongly connected components

If there are many context-action pairs that cause changes, it is not particularly useful to introduce an option for each of them. Instead, the VISA algorithm merges two strongly connected components that are linked by too many exits. After the VISA algorithm identifies exits for a strongly connected component, the algorithm

counts the number of exits identified. If the number of exits is larger than half the total number of state-option pairs of a parent strongly connected component, those two strongly connected components are merged. The merge operation places all state variables in both strongly connected components into a single component, and recomputes the exits of the new component. As a result, the complexity of solving an associated subtask increases, since there are more state variables in the set $\mathbf{Z}$. However, the number of subtasks decreases since there are fewer exits as a result.

4.1.11 Summary of the algorithm

At this point, we feel it is appropriate to summarize the VISA algorithm. The many steps of the algorithm make it difficult to follow. An overview of the VISA algorithm is given in Algorithm 1.

Algorithm 1 The VISA algorithm

```
1: Input: DBN model of a factored MDP 𝓜 with set of state variables S
2: construct the causal graph of the task
3: compute the strongly connected components of the causal graph
4: perform a topological sort of the strongly connected components
5: for each strongly connected component SC ⊆ S in topological order
6:     identify exits that cause the values of state variables in SC to change
7:     while the number of exits exceeds a threshold
8:         merge SC with a parent strongly connected component
9:         label the resulting strongly connected component SC and recompute the exits
10:    for each exit ⟨c, a⟩ of the strongly connected component SC
11:        perform any possible exit transformations
12:        compute the set Z of influencing state variables
13:        construct an initiation set I
14:        construct a termination function β using the context c
15:        construct a policy tree by merging transition graphs of parent components
16:        let S_o be the leaves of the policy tree
17:        let O_o be the set of options that cause state variable changes in Z
18:        let Ψ_o be defined by the initiation sets of options in O_o
19:        define R_o as −1 everywhere except when the context c is unreachable
20:        construct the option SMDP 𝓜_o = ⟨S_o, O_o, Ψ_o, P_o, R_o⟩, with P_o undefined
21:        construct an exit option o = ⟨I, π, β⟩, where π is the optimal policy of 𝓜_o
22:    compute the transition graph of the strongly connected component SC
23: use SMDP Q-learning to learn the policies of each exit option
```

4.1.12 Limitations of the algorithm

The VISA algorithm only decomposes a task if there are two or more strongly connected components in the causal graph of the task. Otherwise, the VISA algorithm cannot exploit conditional independence between state variables to identify options. Since the option SMDPs are stand-alone, the hierarchy discovered by the VISA algorithm enables recursive optimality at best, as opposed to hierarchical optimality (Dietterich, 2000a). In addition, the VISA algorithm works best when there are relatively few exits that cause the values of state variables in a strongly connected component to change.

Furthermore, the option-specific state abstraction performed by the VISA algorithm is independent of the way options are formed. Given access to the causal graph, the VISA algorithm makes it possible to efficiently perform state abstraction for any option whose goal is to reach a context specified by an assignment of values to a subset of the state variables. For the purpose of state abstraction, it does not matter how an autonomous agent determines that it is useful to reach that specific context. In other words, the state abstraction part of the VISA algorithm could be combined with other techniques for discovering useful activities, as long as they are of the required form.

4.2 Experimental results

We ran several experiments to test the empirical performance of the VISA algorithm. Since the VISA algorithm assumes that the DBN model of factored MDPs is given prior to learning, it would be unfair to compare it to algorithms that assume less prior knowledge. Instead, we compared the VISA algorithm to two algorithms that also assume knowledge of the DBN model: Structured Policy Iteration, or SPI (Boutilier et al., 1995), and symbolic Real-Time Dynamic Programming, or sRTDP (Feng et al., 2003). SPI is a more efficient version of policy iteration that takes ad-

vantage of the compactness of the DBN model to approximate the value function in the form of a tree.

sRTDP is an online planning algorithm that, at each step, constructs a set of states that are similar to the current state according to one of two heuristics, value and reach. The algorithm uses the DBN model to determine the set of possible next states, and performs a masked backup of the value function restricted to the set of current and next states. The algorithm then selects for execution one of the actions whose current action-value estimate is highest. sRTDP stores the value function in the form of an algebraic decision diagram, or ADD, and uses the SPUDD algorithm (Hoey et al., 1999) to perform the masked value backup at each step. The SPUDD algorithm includes a mechanism that limits the size of the ADDs, divides the state variables into subsets, and decomposes the value backup into several smaller computations. In our implementation, we did not allow the size of the ADDs to exceed 10,000 nodes.

We performed experiments with each algorithm in four tasks: the coffee task, the Taxi task (Dietterich, 2000a), the Factory task (Hoey et al., 1999), and a simplified version of the autonomous guided vehicle (AGV) task of Ghavamzadeh and Mahadevan (2001). The Taxi task is described in Chapter 3. In the Factory task (Hoey et al., 1999), a robot has to assemble a component made of two objects. Before assembly is possible, the robot has to perform various operations on each of the two objects, such as shaping, smoothing, polishing and painting. The objects can then be connected either by drilling and bolting or by glueing. The task is described by 17 binary variables for a total of 130,000 states, and the robot has 14 actions.

In the AGV task (Ghavamzadeh and Mahadevan, 2001), an autonomous guided vehicle (AGV) has to transport parts between machines in a manufactoring workshop. We simplified the task by reducing the number of machines from 4 to 2 and setting the processing time of machines to 0 to make the task fully observable. The resulting task is illustrated in Figure 4.6 and has approximately 75,000 states. Even though we

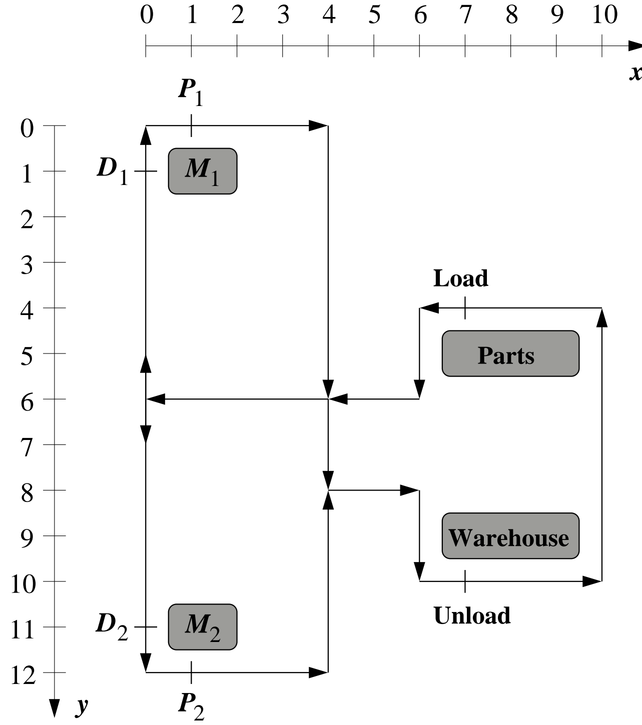

Figure 4.6. Illustration of the AGV task

simplified the AGV task, it is still much larger than any task that we know of for which discovery of activities has been successfully attempted. The goal of the AGV is to proceed to the load station, pick up a random part i, transport it to the drop-off location D_i of machine M_i, drop it off, then proceed to the pick-up location P_i of machine M_i, pick up the processed part, transport it to the unload station, and finally drop it off. The AGV is restricted to move unidirectionally along the arrows in the figure, and has to ensure that at least one part of each type is stored in the warehouse. The set of state variables describing the task is $\mathbf{S} = \{S_x, S_y, S_f, S_h, S_{d1}, S_{p1}, S_{d2}, S_{p2}, S_{a1}, S_{a2}\}$, where S_x and S_y represent the location of the AGV, S_f the direction it is facing, S_h the part it is holding, S_{di} the number of parts at the drop-off location D_i of machine M_i, S_{pi} the number of parts at the pick-up location P_i, and S_{ai} whether a part of type i is present in the warehouse. The AGV has 6 actions: move in the direction it is facing, turn left or right, drop off a part, pick up a part, and idle.

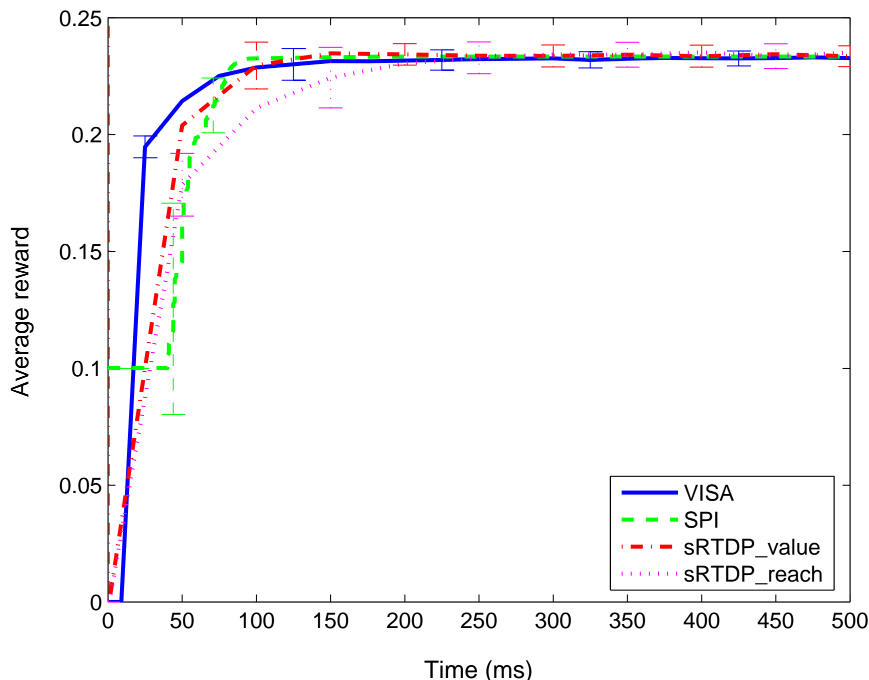

Figure 4.7. Results of learning in the coffee task

Each graph that follows illustrates the average reward over 100 learning runs with each algorithm. Since the algorithms are fundamentally different we compared the actual running time in milliseconds. The graphs for the VISA algorithm include the time it takes to decompose the factored MDP. We used SMDP Q-learning to learn the option policies, which reduces to regular Q-learning for policies that select between primitive actions. Prior to executing, sRTDP computes action ADDs; the graphs include the time it takes to do this. We report results of both heuristics, value and reach, used by sRTDP to construct the set of similar states. All algorithms were coded in Java, except that the CUDD library (written in C) was used to manipulate ADDs through the Java Native Interface.

Figure 4.7 illustrates the results of the experiments in the coffee task. The decomposition discovered by the VISA algorithm uses a total of 26 state-option pairs to represent the option policies. In comparison, the total number of state-action pairs

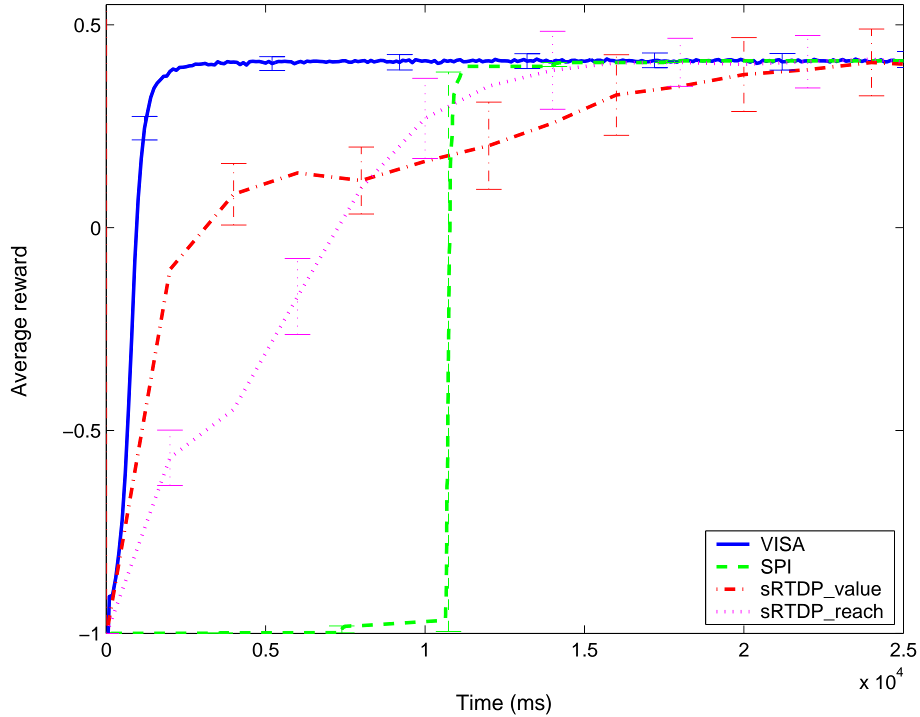

Figure 4.8. Results of learning in the Taxi task

of the original task is 256. We believe this is the reason that learning converges faster using the hierarchy discovered by VISA. However, the VISA algorithm performs only marginally better than the other algorithms in the coffee task.

Figure 4.8 illustrates the results of the experiments in the Taxi task. In the Taxi task, the VISA algorithm performs significantly better than the other algorithms. The reason that the VISA algorithm outperforms the other algorithms is that VISA decomposes the task into smaller, stand-alone tasks that are easier to solve without ever enumerating the entire state space. In the Taxi task, the VISA algorithm reduces the number of state-option pairs from 3,000 in the original task to approximately 800 in the decomposition. In addition, the options introduced by VISA facilitate exploration by providing subgoals that direct the taxi towards the ultimate goal of delivering the passenger.

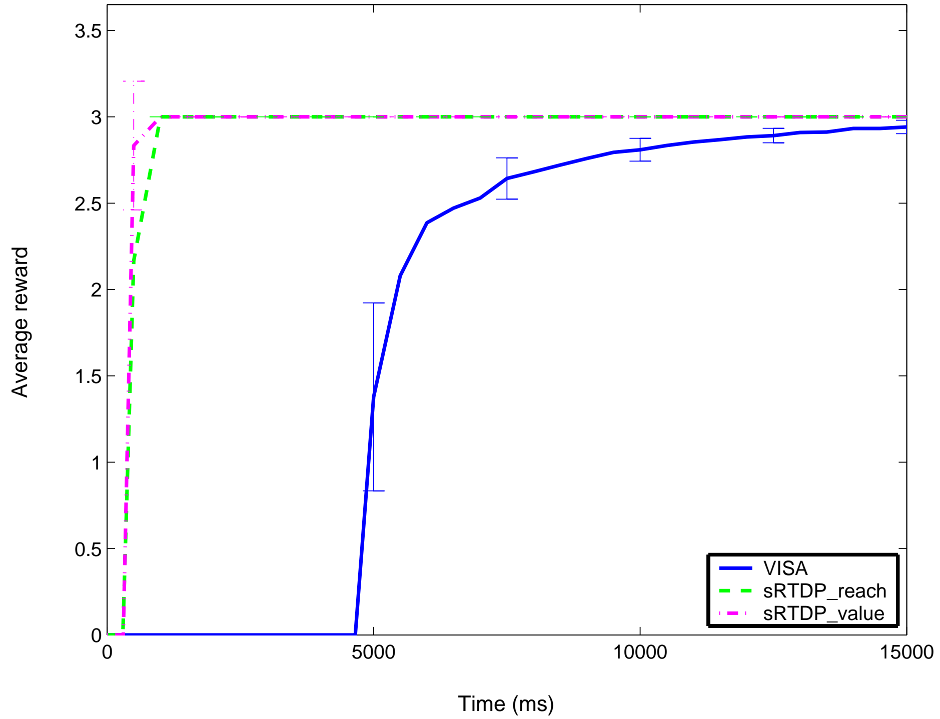

Figure 4.9. Results of learning in the Factory task

Figure 4.9 illustrates the results of the experiments in the Factory task of the VISA algorithm and sRTDP. The VISA algorithm decomposes the task in 5 seconds and learning converges after 20 seconds. SPI ran out of memory after running for several hours. sRTDP converges after approximately 1 second, and thus outperforms the VISA algorithm in this task. We believe that the reason sRTDP performs well in this task is that the solution path is relatively short (5-10 steps) and that intermediate reward is provided along the way. sRTDP is thus able to focus quickly on states that lead to the goal, and never has to perform value backups for irrelevant states. Since the VISA algorithm uses reinforcement learning to learn the policy of the options, it is more likely to get sidetracked through exploration, causing learning to be somewhat slower. Since the solution path is already short, hierarchical decomposition has a less positive impact on exploration.

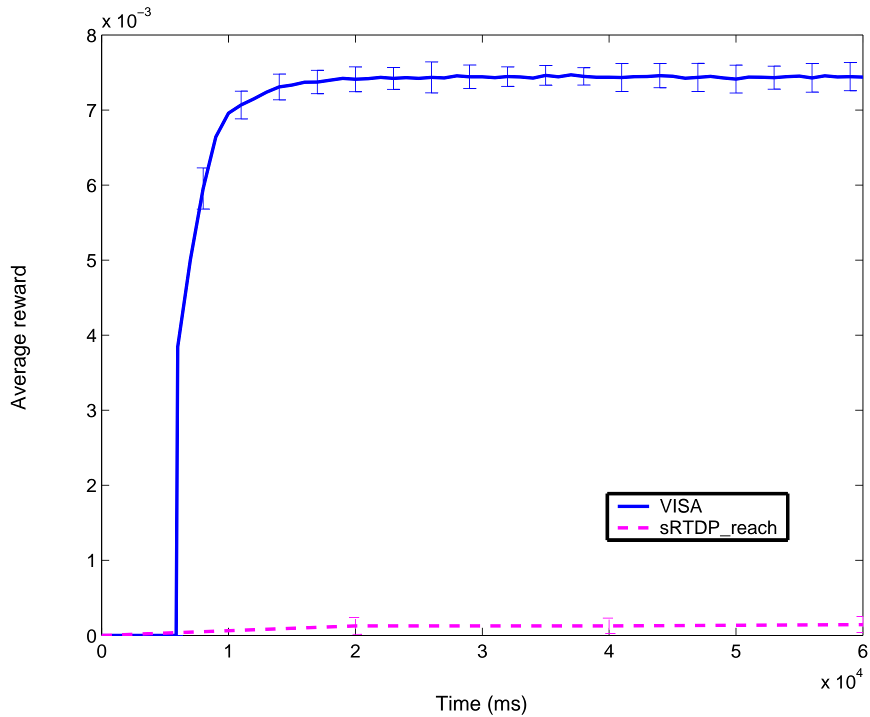

Figure 4.10. Results of learning in the AGV task

Figure 4.10 illustrates the results of the experiments in the AGV task of the VISA algorithm and sRTDP using the reach heuristic. In this case, the VISA algorithm reduces the number of state-option pairs from 450,000 to approximately 16,000. The VISA algorithm decomposes the task in roughly 6 seconds and learning converges after 20 seconds. In comparison, SPI ran out of memory after 3 hours. sRTDP using the reach heuristic completes the task a few times within the first minute of running time but convergence is much slower than VISA. During our experiments, sRTDP using the value heuristic failed to complete the task even once within the first 15 minutes. The AGV task has a much longer solution path than the Factory task (approximately 100 steps) and reward is only provided upon completion of the task. To correctly propagate values throughout the state space, sRTDP has to complete the task many times. In contrast, the options introduced by VISA guide the AGV towards

completing the task and help update the value more quickly. Selecting between options, the solution path only requires 8 steps.

The results of the experiments illustrate the power of hierarchical decomposition when combined with option-specific abstraction. Even though SPI and sRTDP take advantage of task structure and are empirically faster than regular reinforcement learning algorithms, they still suffer from the curse of dimensionality as the size of the state space grows. On the other hand, the VISA algorithm decomposes the original tasks into smaller, stand-alone tasks that are easier to solve without ever enumerating the state space. Instead, the complexity of the decomposition is polynomial in the size of the conditional probability trees of the DBN model. Each stand-alone task only distinguishes between values of a subset of the state variables, which means that the complexity of learning does not necessarily increase with the number of state variables.

4.3 Discussion

It is possible to combine the VISA algorithm with several of the other techniques for scaling reinforcement learning. For example, once the VISA algorithm has decomposed a task into options, we can apply reinforcement learning with function approximation to learn the option policies. Another possibility is to use existing algorithms to detect bottlenecks in the transition graph of a strongly connected component in the causal graph. This would enable further decomposition of the option SMDPs into even smaller subtasks.

Recall that the VISA algorithm performs state abstraction for an option SMDP by constructing the partition $\Lambda_{\mathbf{Z}}$ induced by the projection $f_{\mathbf{Z}}$, where $\mathbf{Z} \subseteq \mathbf{S}$ is the set of state variables in strongly connected components whose variable values appear in the context of the associated exit. As a result of state abstraction, the option policy may be suboptimal. The problem occurs when an option selected by the option policy

changes the value of a state variable not in **Z** that indirectly influences state variables in **Z**. This problem would be alleviated if we merge strongly connected components whose state variables are affected by the same actions. The resulting decomposition would be less efficient in terms of learning complexity but would guarantee recursive optimality.

4.4 Related work

There exist several other algorithms that decompose tasks into a hierarchy of activities, one of which is the HEX-Q algorithm (Hengst, 2002) already mentioned. Nested Q-learning (Digney, 1996) introduces an activity for each value of each state variable. The goal of each activity is to reach the context described by the single state variable value. McGovern and Barto (2001) use diverse density to locate bottlenecks in successful solution paths, and introduce activities that reach these bottlenecks. Şimşek and Barto (2004) measure the relative novelty of each visited state, and introduce activities that reach states whose relative novelty exceeds a threshold value. Recent work on intrinsic motivation Barto et al. (2004) tracks salient changes in variable values and introduces activities that cause salient changes to occur.

Other researchers use graph-theoretic approaches to decompose tasks. Menache et al. (2002) construct a state transition graph and introduce activities that reach states on the border of strongly connected regions of the graph. The authors use a max-flow/min-cut algorithm to identify border states in the transition graph. Mannor et al. (2004) use a clustering algorithm to partition the state space into different regions and introduce activities for moving between regions. Şimşek et al. (2005) identify subgoals by partitioning local state transition graphs that represent only the most recently recorded experience. Bulitko et al. (2005) decompose deterministic tasks by repeatedly clustering states into abstract states and constructing a new

transition graph for the abstract states. At each level of abstraction, the authors introduce activities that move between abstract states.

Another approach is to track learning in several related tasks and identify activities that are useful across tasks. SKILLS (Thrun and Schwartz, 1996) identifies activities that minimize a function of the performance loss induced by the resulting hierarchy and the total description length of all actions. PolicyBlocks (Pickett and Barto, 2002) identifies regions in the state space for which the policy is identical across tasks, and introduces activities that represent the policy of each region. Each activity is only admissible within its region of the state space.

Helmert (2004) developed an algorithm that constructs a causal graph similar to that of the VISA algorithm and uses the graph to decompose deterministic planning tasks. The algorithm assumes a STRIPS formulation of actions (Fikes and Nilsson, 1971), which is similar to the DBN model of factored MDPs. Just like the DBN model, the STRIPS formulation expresses actions in terms of causes and effects on the state variables, except that the causes and effects are deterministic. Helmert (2004) uses the STRIPS action formulation to construct a causal graph in a special class of deterministic tasks, in which the causal graph has one absorbing state variable with edges from each other state variable. The author shows that his algorithm efficiently solves a set of standard planning tasks using activities to represent the stand-alone tasks of the resulting decomposition.

The VISA algorithm presented in this chapter is based on the assumption that the values of key state variables change relatively infrequently. This is the same assumption made by Hengst (2002), Helmert (2004), and Barto et al. (2004). Just like the HEX-Q algorithm (Hengst, 2002), the VISA algorithm decomposes a task into activities by detecting the combinations of state variable values and actions that cause key variable value changes. However, as we already discussed, the VISA algorithm uses the causal graph to represent how state variables are related, which is a more

realistic model than that used by HEX-Q. Unlike intrinsic motivation (Barto et al., 2004), the VISA algorithm does not need to designate certain variable value changes as salient. Unlike the work of Helmert (2004), the VISA algorithm can handle any configuration of the causal graph.

Most other existing algorithms need to accumulate extensive experience in the environment to decompose a task into activities, and usually store quantities for each state. Assuming that the DBN model is given, the VISA algorithm does not need to accumulate experience in the environment to perform the decomposition. In addition, the VISA algorithm only stores quantities proportional to the size of the conditional probability trees of the DBN model. Although we do not provide any comparisons, it is likely that the VISA algorithm uses less memory and performs decomposition of a task in less time than these other algorithms. Naturally, the assumption that key variable value changes occur relatively infrequently may not always hold, and the DBN model may not always be given. We believe that these are the two main drawbacks of our algorithm.

There are several efficient algorithms for solving factored MDPs that use the DBN model to compactly describe transition probabilities and expected reward. Structured Policy Iteration, or SPI (Boutilier et al., 1995), stores the policy and value function in the form of trees. The algorithm performs policy iteration by intermittently updating the policy and value function, possibly changing the structure of the trees in the process. Hoey et al. (1999) modified SPI to include algebraic decision diagrams, or ADDs, which store conditional probabilities more compactly than trees. Symbolic Real-Time Dynamic Programming, or sRTDP (Feng et al., 2003), also assumes that the conditional probabilities of the DBN model are stored using ADDs. The algorithm clusters states into abstract states based on two criteria, and performs an efficient backup of the value of the current abstract state following each execution of an action in the environment.

The DBN-E^3 algorithm (Kearns and Koller, 1999) assumes that there exists an approximate planning algorithm for the task, and that the structure of the DBN model is given. Using the planning procedure as a subroutine, the algorithm explores the state space and fills in the parameters of the DBN model. The running time of the algorithm is polynomial in the number of parameters of the DBN model, generally much smaller than the number of states. Guestrin et al. (2001) developed an algorithm based on linear programming that combines the DBN model with max-norm projections to solve factored MDPs. The algorithm assumes that there is a set of basis functions for representing the value function and is guaranteed to converge to an approximately optimal solution.

CHAPTER 5

CONSTRUCTING COMPACT OPTION MODELS

In the previous chapter, we devised an algorithm, the VISA algorithm, that uses the DBN model of factored MDPs to decompose tasks into hierarchies of activities. The VISA algorithm identifies exits, pairs of a context and an action that causes the value of a state variable to change, and introduces an exit option associated with each exit. For each exit option, the VISA algorithm constructs an option SMDP that implicitly represents the option policy. Since the VISA algorithm does not have access to an estimate of the transition probabilities of the option SMDP, it cannot use planning algorithms to solve the option SMDP. Instead, the VISA algorithm uses SMDP Q-learning, which does not require knowledge of the transition probabilities, to learn the policies of the options.

As we already discussed, there exist several efficient algorithms for solving factored MDPs which assume that a DBN model is given. These algorithms take advantage of the compact structure inherent in the DBN model to construct efficient solutions. If the VISA algorithm had access to DBN models that compactly describe the effect of options, it would be possible to apply these existing algorithms to efficiently solve the option SMDPs of the exit options. Access to DBN models of options would open up new possibilities for learning and planning with options. Unfortunately, there are currently no techniques that compute compact models of activities without enumerating the state space. Since the goal of the VISA algorithm is to alleviate the curse of dimensionality, we want to avoid enumerating the state space if possible.

In this chapter, we devise an algorithm for computing compact option models without enumerating the state space. Our work combines the compactness of the DBN model with the simplification offered by hierarchical decomposition. Once an option has been learned, it can be cached and added to the set of actions for subsequent learning and planning. We enhance the description of a learned option by constructing a compact representation of its effect on the state variables, making it possible to learn and plan with options using the more efficient algorithms that take advantage of compact representations.

This chapter makes several contributions. First, we analyze the complexity of constructing a model that makes it possible to treat a learned option as a single unit during learning and planning. We investigate how to reduce the complexity through the use of partitions with certain properties. Finally, we show how to construct partitions with the required properties for exit options discovered by the VISA algorithm. To construct partitions, we develop novel operations on decision trees.

5.1 Multi-time option models

Sutton et al. (1999) defined the multi-time model of an option $o = \langle I, \pi, \beta \rangle$ as

$$P(s' \mid s, o) = \sum_{t=1}^{\infty} \gamma^t P(s', t \mid s, o), \tag{5.1}$$

$$R(s, o) = E\{\sum_{k=1}^{t} \gamma^{k-1} R(s_k, a_k) \mid s_1 = s\}, \tag{5.2}$$

where t is the number of time steps until o terminates, and $P(s', t \mid s, o)$ is the probability that o terminates in state s' after t time steps when executed in state s. The expectation in the expression for $R(s, o)$ is taken over the distribution of state-action pairs (s_k, a_k), $k \in [1, t]$. This distribution is determined by the functions $P(s_{k+1} \mid s_k, a_k)$, $\pi(s_k, a_k)$, and $\beta(s_k)$. The terms $P(s' \mid s, o)$ are not true probabilities since they do not sum to 1 for $\gamma < 1$. However, the multi-time model enables learning

and planning with options as single units, which Sutton et al. (1999) call SMDP value learning and SMDP planning, respectively.

It is possible to use dynamic programming to compute the multi-time model in Equations (5.1) and (5.2). We can set up the Bellman form of the equations in which each term is a function of the terms at the next time step:

$$P(s' \mid s,o) = \gamma \sum_{a \in A} \pi(s,a) \left[P(s' \mid s,a)\beta(s') + \sum_{s'' \in S} P(s'' \mid s,a)(1-\beta(s''))P(s' \mid s'',o) \right], \tag{5.3}$$

$$R(s,o) = \sum_{a \in A} \pi(s,a) \left[R(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a)(1-\beta(s'))R(s',o) \right]. \tag{5.4}$$

Let us label each state with a unique subscript $i \in \{1, \ldots, |S|\}$. Let P^a, $a \in A$, be the transition matrix for action a whose entry (i,j) equals $P(s_j \mid s_i, a)$, and let P^o be the corresponding matrix for option o. Let Π^a, $a \in A$, be the diagonal matrix whose entry (i,i) equals $\pi(s_i,a)$, and let B be the diagonal matrix whose entry (i,i) equals $\beta(s_i)$. Let R^a, $a \in A$, be the vector whose ith entry equals $R(s_i,a)$, and let R^o be the corresponding vector for option o. To avoid confusion with the option initiation set I, we use E to denote the identity matrix. Then we can write the equations for the multi-time model of option o in matrix form as

$$P^o = \gamma \sum_{a \in A} \Pi^a P^a (B + (E-B)P^o), \tag{5.5}$$

$$R^o = \sum_{a \in A} \Pi^a (R^a + \gamma P^a (E-B) R^o). \tag{5.6}$$

The unknown quantities that we want to solve for are P^o and R^o. If we move the unknowns to the left-hand side of the equations we obtain the following system of equations:

$$\left[E - \gamma \sum_{a \in A} \Pi^a P^a (E-B) \right] P^o = \gamma \sum_{a \in A} \Pi^a P^a B, \tag{5.7}$$

$$\left[E - \gamma \sum_{a \in A} \Pi^a P^a (E-B) \right] R^o = \sum_{a \in A} \Pi^a R^a. \tag{5.8}$$

Definition 5.1.1 *An option o is proper if for each state $s_i \in I$, o eventually terminates with probability 1.*

Definition 5.1.1 imposes a restriction on the policy π and termination condition function β of an option o.

Theorem 5.1.2 *For a proper option o, the systems of linear equations in (5.7) and (5.8) are consistent and have unique solutions.*

Note that the unknown quantities P^o and R^o are multiplied by the same matrix $M = \left[E - \gamma \sum_{a \in A} \Pi^a P^a (E - B)\right]$. The systems of linear equations in (5.7) and (5.8) are consistent and have unique solutions if and only if matrix M is invertible, i.e., if the determinant of M is non-zero. The complete proof of Theorem 5.1.2 appears in Appendix A.

The complexity of computing the multi-time model of an option o is comparable to the complexity of computing an optimal policy of an MDP using dynamic programming. If it is possible to compute an optimal policy in the time it takes to compute the multi-time model of a single option, there is little need to decompose the task into options in the first place. It follows that computing the multi-time model of an option o is comparatively costly. To efficiently compute compact models of the transition probabilities and expected reward of options, we would like to develop techniques for reducing the complexity.

5.2 Options in factored MDPs

Recall that in a factored MDP, it is possible to approximate the transition probabilities as products of the conditional probabilities of each state variable $S_d \in \mathbf{S}$:

$$P(\mathbf{s}' \mid \mathbf{s}, a) \approx \prod_{S_d \in \mathbf{S}} P_d(f_{\{S_d\}}(\mathbf{s}') \mid f_{\mathbf{Pa}(S_d)}(\mathbf{s}), a).$$

It is possible to approximate the terms of the multi-time model in a similar way. However, the multi-time model of an option o has two distributions that resemble transition probabilities: $P(\mathbf{s}' \mid \mathbf{s}, o)$, the discounted probability of transitioning from $\mathbf{s}$ to $\mathbf{s}'$ as a result of executing o; and $P(\mathbf{s}', t \mid \mathbf{s}, o)$, the exact probability of transitioning from $\mathbf{s}$ to $\mathbf{s}'$ in t time steps as a result of executing o. We can choose which of the two distributions to approximate.

If we choose to approximate $P(\mathbf{s}' \mid \mathbf{s}, o)$, we obtain the following approximation:

$$P(\mathbf{s}' \mid \mathbf{s}, o) \approx \prod_{S_d \in \mathbf{S}} P_d(f_{\{S_d\}}(\mathbf{s}') \mid f_{\mathbf{Pa}(S_d)}(\mathbf{s}), o). \tag{5.9}$$

Since we do not (yet) have access to a DBN model of option o, we assume that all state variables are parents of S_d, so $f_{\mathbf{Pa}(S_d)}(\mathbf{s}) = f_{\mathbf{S}}(\mathbf{s}) = \mathbf{s}$. We can compute the terms $P_d(f_{\{S_d\}}(\mathbf{s}') \mid \mathbf{s}, o)$ in the same way as the multi-time model:

$$P_d(f_{\{S_d\}}(\mathbf{s}') \mid \mathbf{s}, o) = \sum_{t=1}^{\infty} \gamma^t P_d(f_{\{S_d\}}(\mathbf{s}'), t \mid \mathbf{s}, o). \tag{5.10}$$

As a result, we obtain the following final approximation of Equation (5.1):

$$P(\mathbf{s}' \mid \mathbf{s}, o) \approx \prod_{S_d \in \mathbf{S}} \sum_{t=1}^{\infty} \gamma^t P_d(f_{\{S_d\}}(\mathbf{s}'), t \mid \mathbf{s}, o). \tag{5.11}$$

The approximation in Equation (5.11) enables us to compute the conditional probabilities $P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}, o)$ of the multi-time model separately for each state variable S_d. Here, $\mathbf{c}_{\{d\}}$ denotes a context described by one of the values of state variable S_d.

If we instead choose to approximate $P(\mathbf{s}', t \mid \mathbf{s}, o)$, we obtain the following alternative approximation of Equation (5.1):

$$P(\mathbf{s}' \mid \mathbf{s}, o) = \sum_{t=1}^{\infty} \gamma^t P(\mathbf{s}', t \mid \mathbf{s}, o) \approx \sum_{t=1}^{\infty} \prod_{S_d \in \mathbf{S}} \gamma^t P_d(f_{\{S_d\}}(\mathbf{s}'), t \mid \mathbf{s}, o). \tag{5.12}$$

Note that the difference between Equations (5.11) and (5.12) is the order of the summation and the product. As a result, Equation (5.11) assigns non-zero probability to events that could never occur, such as "the value of state variable $S_{\mathtt{L}}$ becomes L in 2 time steps, and the value of state variable $S_{\mathtt{W}}$ becomes W in 3 time steps." This event could never occur since an option could not simultaneously terminate after 2 time steps and 3 time steps. In this sense, Equation (5.12) is a better approximation of $P(\mathbf{s}' \mid \mathbf{s}, o)$, but on the other hand, it does not enable us to compute a multi-time model of option o separately for each state variable. As we shall see, the ability to decompose the computation significantly reduces the complexity of computing the multi-time model. We believe that the reduction in complexity justifies the loss of accuracy, although we currently have no bounds on the approximation error. For this reason, we will use Equation (5.11) as our approximation of Equation (5.1).

For each state variable S_d, each state $\mathbf{s} \in S$, and each context $\mathbf{c}_{\{d\}}$, we want to compute the term $P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}, o)$, given by

$$P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}, o) = \gamma \sum_{a \in A} \pi(\mathbf{s}, a) \sum_{\mathbf{s}' \in S} P(\mathbf{s}' \mid \mathbf{s}, a) \left[\beta(\mathbf{s}') \delta_{\mathbf{c}_{\{d\}}, f_{\{S_d\}}(\mathbf{s}')} + (1 - \beta(\mathbf{s}')) P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}', o) \right], \tag{5.13}$$

where $\delta_{i,j}$ is Kronecker's delta. Let P_d^o be the transition matrix for option o and state variable S_d whose entry (i, j) equals $P_d(\mathbf{c}_{\{d\}} = j \mid \mathbf{s}_i, o)$. Let F_d be the matrix whose entry (i, j) equals 1 if $f_{\{S_d\}}(\mathbf{s}_i) = j$, and 0 otherwise. In other words, P_d^o and F_d are $|S| \times |Val(S_d)|$ matrices. Then we can write (5.13) as

$$P_d^o = \gamma \sum_{a \in A} \Pi^a P^a (B F_d + (E - B) P_d^o). \tag{5.14}$$

Let us again move all unknowns to the left side of the equation to obtain

$$\left[E - \gamma \sum_{a \in A} \Pi^a P^a (E - B) \right] P_d^o = \gamma \sum_{a \in A} \Pi^a P^a B F_d. \tag{5.15}$$

Lemma 5.2.1 *For a proper option o, the system of linear equations in (5.15) is consistent and has a unique solution.*

The proof of Lemma 5.2.1 follows directly from the proof of Theorem 5.1.2 since the matrix $M = \left[E - \gamma \sum_{a \in A} \Pi^a P^a (E - B)\right]$ that we need to invert to solve (5.15) is the same as the matrix in (5.7) and (5.8).

5.3 Partitions

Recall that a partition Λ of the state set S that has the stochastic substitution property and is reward respecting induces a reduced MDP that preserves optimality. We define three more properties of partitions of S with respect to a factored MDP $\mathcal{M}$ and an option o:

Definition 5.3.1 *A partition Λ of S is policy respecting if for each pair of states $(\mathbf{s}_i, \mathbf{s}_j) \in S^2$ and each action $a \in A$, $[\mathbf{s}_i]_\Lambda = [\mathbf{s}_j]_\Lambda$ implies that $\pi(\mathbf{s}_i, a) = \pi(\mathbf{s}_j, a)$.*

Definition 5.3.2 *A partition Λ of S is termination respecting if for each pair of states $(\mathbf{s}_i, \mathbf{s}_j) \in S^2$, $[\mathbf{s}_i]_\Lambda = [\mathbf{s}_j]_\Lambda$ implies that $\beta(\mathbf{s}_i) = \beta(\mathbf{s}_j)$.*

Definition 5.3.3 *A partition Λ of S respects a state variable S_d if for each pair of states $(\mathbf{s}_i, \mathbf{s}_j) \in S^2$, $[\mathbf{s}_i]_\Lambda = [\mathbf{s}_j]_\Lambda$ implies that $f_{\{S_d\}}(\mathbf{s}_i) = f_{\{S_d\}}(\mathbf{s}_j)$.*

Using these definitions, it is possible to construct partitions that simplify computation of the multi-time model of an option o, which we prove in the following two lemmas:

Lemma 5.3.4 *Let o be a proper option and let Λ_d be a partition that has the stochastic substitution property, is policy respecting, termination respecting, and respects state variable S_d. Then for each pair of states $(\mathbf{s}_i, \mathbf{s}_j) \in S^2$ and each context $\mathbf{c}_{\{d\}}$, $[\mathbf{s}_i]_{\Lambda_d} = [\mathbf{s}_j]_{\Lambda_d}$ implies that $P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}_i, o) = P_d(\mathbf{c}_{\{d\}} \mid s_j, o)$.*

Proof Since Λ_d is termination respecting, the termination condition $\beta(\mathbf{s}_k)$ is equal for states $\mathbf{s}_k \in \lambda$ in block λ of partition Λ_d. Let β_λ denote the common termination condition $\beta(\mathbf{s}_k)$ of states $\mathbf{s}_k \in \lambda$. Since Λ_d respects state variable S_d, the projection $f_{\{S_d\}}(\mathbf{s}_k)$ is equal for states $\mathbf{s}_k \in \lambda$ in block λ of partition Λ_d. Let f_λ denote the common projection $f_{\{S_d\}}(\mathbf{s}_k)$ of states $\mathbf{s}_k \in \lambda$. Assume that for each block λ, each state $\mathbf{s}_k \in \lambda$, and each context $\mathbf{c}_{\{d\}}$, the probability $P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}_k, o)$ is equal, and let $P^o_{\lambda,\mathbf{c}_{\{d\}}}$ denote that probability. We will show that $P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}_i, o) = P_d(\mathbf{c}_{\{d\}} \mid s_j, o)$ checks under this assumption if $[\mathbf{s}_i]_{\Lambda_d} = [\mathbf{s}_j]_{\Lambda_d}$.

From Equation (5.13), the expression for $P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}_i, o)$ is given by

$$\gamma \sum_{a \in A} \pi(\mathbf{s}_i, a) \sum_{\mathbf{s}' \in S} P(\mathbf{s}' \mid \mathbf{s}_i, a) \left[\beta(\mathbf{s}')\delta_{\mathbf{c}_{\{d\}}, f_{\{S_d\}}(\mathbf{s}')} + (1 - \beta(\mathbf{s}'))P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}', o)\right].$$

We can expand the sum $\sum_{\mathbf{s}' \in S}$ by first summing over blocks λ of partition Λ_d and then over states $\mathbf{s}_k$ in block λ:

$$\gamma \sum_{a \in A} \pi(\mathbf{s}_i, a) \sum_{\lambda \in \Lambda_d} \sum_{\mathbf{s}_k \in \lambda} P(\mathbf{s}_k \mid \mathbf{s}_i, a) \left[\beta(\mathbf{s}_k)\delta_{\mathbf{c}_{\{d\}}, f_{\{S_d\}}(\mathbf{s}_k)} + (1 - \beta(\mathbf{s}_k))P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}_k, o)\right] =$$
$$= \gamma \sum_{a \in A} \pi(\mathbf{s}_i, a) \sum_{\lambda \in \Lambda_d} \sum_{\mathbf{s}_k \in \lambda} P(\mathbf{s}_k \mid \mathbf{s}_i, a) \left[\beta_\lambda \delta_{\mathbf{c}_{\{d\}}, f_\lambda} + (1 - \beta_\lambda) P^o_{\lambda,\mathbf{c}_{\{d\}}}\right].$$

Since the terms within the parentheses do not depend on $\mathbf{s}_k$, we can move them outside the summation over $\mathbf{s}_k$ to obtain

$$P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}_i, o) = \gamma \sum_{a \in A} \pi(\mathbf{s}_i, a) \sum_{\lambda \in \Lambda_d} \left[\beta_\lambda \delta_{\mathbf{c}_{\{d\}}, f_\lambda} + (1 - \beta_\lambda) P^o_{\lambda,\mathbf{c}_{\{d\}}}\right] \sum_{\mathbf{s}_k \in \lambda} P(\mathbf{s}_k \mid \mathbf{s}_i, a).$$

We can expand the expression for $P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}_j, o)$ in the same way to obtain

$$P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}_j, o) = \gamma \sum_{a \in A} \pi(\mathbf{s}_j, a) \sum_{\lambda \in \Lambda_d} \left[\beta_\lambda \delta_{\mathbf{c}_{\{d\}}, f_\lambda} + (1 - \beta_\lambda) P^o_{\lambda,\mathbf{c}_{\{d\}}}\right] \sum_{\mathbf{s}_k \in \lambda} P(\mathbf{s}_k \mid \mathbf{s}_j, a).$$

Since Λ_d has the stochastic substitution property, $[\mathbf{s}_i]_{\Lambda_d} = [\mathbf{s}_j]_{\Lambda_d}$ implies that that $\sum_{\mathbf{s}_k \in \lambda} P(\mathbf{s}_k \mid \mathbf{s}_i, a) = \sum_{\mathbf{s}_k \in \lambda} P(\mathbf{s}_k \mid \mathbf{s}_j, a)$ for each action $a \in A$ and each block λ

of partition Λ_d. Since Λ_d is policy respecting, $[\mathbf{s}_i]_{\Lambda_d} = [\mathbf{s}_j]_{\Lambda_d}$ implies that $\pi(\mathbf{s}_i, a) = \pi(\mathbf{s}_j, a)$ for each action $a \in A$. It follows that $P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}_i, o) = P_d(\mathbf{c}_{\{d\}} \mid s_j, o)$ checks under the assumption that we made above. Lemma 5.2.1 states that the solution to the equations in (5.15) is unique. Since we know that $P_d(\mathbf{c}_{\{d\}} \mid \mathbf{s}_i, o) = P_d(\mathbf{c}_{\{d\}} \mid s_j, o)$ is a solution, it follows from Lemma 5.2.1 that it is the only solution. This concludes the proof.

Because of the properties of Λ_d, each matrix in Equation (5.15) corresponds to a reduced, equivalent matrix, which we denote using subscript Λ_d. Instead of one row or column per state $s_i \in S$, the reduced matrix has one row or column per block $\lambda_i \in \Lambda_d$. The reduced system of linear equations for state variable S_d and option o is given by

$$\left[E - \gamma \sum_{a \in A} \Pi^a_{\Lambda_d} P^a_{\Lambda_d} (E - B_{\Lambda_d})\right] P^o_{\Lambda_d} = \gamma \sum_{a \in A} \Pi^a_{\Lambda_d} P^a_{\Lambda_d} B_{\Lambda_d} F_{\Lambda_d}. \qquad (5.16)$$

Lemma 5.3.5 *Let o be a proper option and let Λ_R be a partition that has the stochastic substitution property, is reward respecting, policy respecting, and termination respecting. Then for each pair of states $(s_i, s_j) \in S^2$, $[s_i]_{\Lambda_R} = [s_j]_{\Lambda_R}$ implies that $R(s_i, o) = R(s_j, o)$.*

Proof Since Λ_R is termination respecting, the termination condition $\beta(\mathbf{s}_k)$ is equal for states $\mathbf{s}_k \in \lambda$ in block λ of partition Λ_R. Let β_λ denote the common termination condition $\beta(\mathbf{s}_k)$ of states $\mathbf{s}_k \in \lambda$. Assume that for each state $\mathbf{s}_k \in \lambda$, the expected reward $R(\mathbf{s}_k, o)$ as a result of executing option o is equal, and let R^o_λ denote that expected reward. We will show that $R(\mathbf{s}_i, o) = R(\mathbf{s}_j, o)$ checks under this assumption if $[\mathbf{s}_i]_{\Lambda_R} = [\mathbf{s}_j]_{\Lambda_R}$.

From Equation (5.4), the expression for $R(\mathbf{s}_i, o)$ is given by

$$R(\mathbf{s}_i, o) = \sum_{a \in A} \pi(\mathbf{s}_i, a) \left[R(\mathbf{s}_i, a) + \gamma \sum_{s' \in S} P(s' \mid \mathbf{s}_i, a)(1 - \beta(s'))R(s', o)\right].$$

We can expand the sum $\sum_{\mathbf{s}' \in S}$ by first summing over blocks λ of partition Λ_R and then over states $\mathbf{s}_k$ in block λ:

$$\begin{aligned} R(\mathbf{s}_i, o) &= \sum_{a \in A} \pi(\mathbf{s}_i, a) \left[R(\mathbf{s}_i, a) + \gamma \sum_{\lambda \in \Lambda_R} \sum_{\mathbf{s}_k \in \lambda} P(\mathbf{s}_k \mid \mathbf{s}_i, a)(1 - \beta(\mathbf{s}_k)) R(\mathbf{s}_k, o) \right] = \\ &= \sum_{a \in A} \pi(\mathbf{s}_i, a) \left[R(\mathbf{s}_i, a) + \gamma \sum_{\lambda \in \Lambda_R} \sum_{\mathbf{s}_k \in \lambda} P(\mathbf{s}_k \mid \mathbf{s}_i, a)(1 - \beta_\lambda) R^o_\lambda \right]. \end{aligned}$$

We move terms that do not depend on $\mathbf{s}_k$ outside the summation over $\mathbf{s}_k$ to obtain

$$R(\mathbf{s}_i, o) = \sum_{a \in A} \pi(\mathbf{s}_i, a) \left[R(\mathbf{s}_i, a) + \gamma \sum_{\lambda \in \Lambda_R} (1 - \beta_\lambda) R^o_\lambda \sum_{\mathbf{s}_k \in \lambda} P(\mathbf{s}_k \mid \mathbf{s}_i, a) \right].$$

We expand the expression for $R(\mathbf{s}_j, o)$ in the same way to obtain

$$R(\mathbf{s}_j, o) = \sum_{a \in A} \pi(\mathbf{s}_j, a) \left[R(\mathbf{s}_j, a) + \gamma \sum_{\lambda \in \Lambda_R} (1 - \beta_\lambda) R^o_\lambda \sum_{\mathbf{s}_k \in \lambda} P(\mathbf{s}_k \mid \mathbf{s}_j, a) \right].$$

Since Λ_R has the stochastic substitution property, $[\mathbf{s}_i]_{\Lambda_R} = [\mathbf{s}_j]_{\Lambda_R}$ implies that that $\sum_{\mathbf{s}_k \in \lambda} P(\mathbf{s}_k \mid \mathbf{s}_i, a) = \sum_{\mathbf{s}_k \in \lambda} P(\mathbf{s}_k \mid \mathbf{s}_j, a)$ for each action $a \in A$ and each block λ of partition Λ_R. Since Λ_R is policy respecting, $[\mathbf{s}_i]_{\Lambda_R} = [\mathbf{s}_j]_{\Lambda_R}$ implies that $\pi(\mathbf{s}_i, a) = \pi(\mathbf{s}_j, a)$ for each action $a \in A$. Since Λ_R is reward respecting, $[\mathbf{s}_i]_{\Lambda_R} = [\mathbf{s}_j]_{\Lambda_R}$ implies that $R(\mathbf{s}_i, a) = R(\mathbf{s}_j, a)$ for each action $a \in A$. It follows that $R(\mathbf{s}_i, o) = R(\mathbf{s}_j, o)$ checks under the assumption that we made above. Theorem 5.1.2 states that the solution to the equations in (5.8) is unique. Since we know that $R(\mathbf{s}_i, o) = R(\mathbf{s}_j, o)$ is a solution, it follows from Theorem 5.1.2 that it is the only solution. This concludes the proof.

Because of the properties of Λ_R, each matrix in Equation (5.8) corresponds to a reduced, equivalent matrix, which we denote using subscript Λ_R. The reduced system of linear equations corresponding to Equation (5.8) is given by

$$\left[E - \gamma \sum_{a \in A} \Pi^a_{\Lambda_R} P^a_{\Lambda_R} (E - B_{\Lambda_R})\right] R^o_{\Lambda_R} = \sum_{a \in A} \Pi^a_{\Lambda_R} R^a_{\Lambda_R}. \tag{5.17}$$

To solve Equation (5.16) we need to invert a $|\Lambda_d| \times |\Lambda_d|$ matrix. To solve Equation (5.17) we need to invert a $|\Lambda_R| \times |\Lambda_R|$ matrix. If the Λ_d and Λ_R are significantly smaller than the set of states S, the partitions result in a considerable reduction in the complexity of computing the multi-time model of option o.

5.4 Finding useful partitions

In the previous section, we showed that partitions can reduce the complexity of computing a multi-time option model in factored MDPs. In this section, we show how to find useful partitions that satisfy the desired properties for options discovered by the VISA algorithm presented in the previous chapter. As in the previous chapter, we will use the coffee task (Boutilier et al., 1995) to illustrate our algorithm. To construct the partitions, we first introduce two novel operations on trees.

5.4.1 Tree operations

We illustrate the tree operations on the tree $\mathcal{T}^{\texttt{GO}}_{\texttt{W}}$ that stores the conditional probabilities of state variable $S_{\texttt{W}}$ as a result of executing action GO in the coffee task. Figure 5.1 illustrates $\mathcal{T}^{\texttt{GO}}_{\texttt{W}}$, which also appears in Figure 2.1. The tree $\mathcal{T}^{\texttt{GO}}_{\texttt{W}}$ induces a partition Λ of S such that two states s_i and s_j belong to the same block of the partition if and only if they map to the same leaf of the tree. Each leaf $\mathcal{L}$ of the tree is associated with a context $\mathbf{c}_{\mathcal{L}}$; for example, the context associated with the left-most leaf of $\mathcal{T}^{\texttt{GO}}_{\texttt{W}}$ is $(S_{\texttt{W}} = W)$.

Definition 5.4.1 *The restriction of a tree $\mathcal{T}$ to a context $\mathbf{c}$, which we denote $\mathcal{T} \mid \mathbf{c}$, is the tree that results from clamping the values of the state variables in the set $\mathbf{C} \subseteq \mathbf{S}$ to $\mathbf{c}$ and collapsing the tree $\mathcal{T}$.*

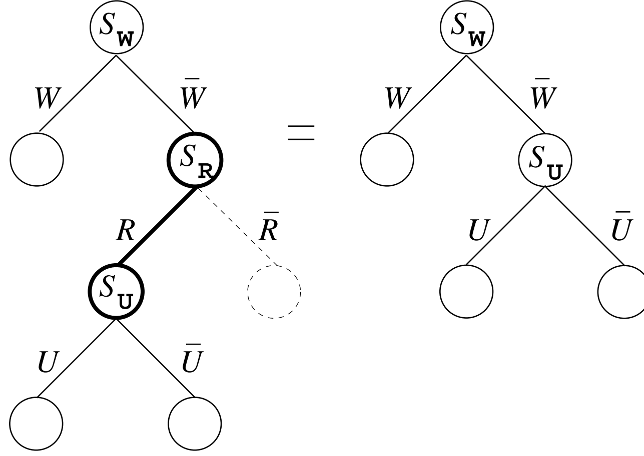

Figure 5.1. The tree $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}}$ and the restriction $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}} \mid (S_{\mathtt{R}} = R)$

Figure 5.1 shows the restriction of $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}}$ to the context $(S_{\mathtt{R}} = R)$. As the figure shows, the restriction clamps the value of state variable $S_{\mathtt{R}}$ to R. Since the value of $S_{\mathtt{R}}$ is known, there is no longer any need to distinguish between values of $S_{\mathtt{R}}$, so we can collapse the tree at the node associated with $S_{\mathtt{R}}$. The subtree rooted at $S_{\mathtt{U}}$ becomes the new child of the root node associated with the value $\overline{W}$ of state variable $S_{\mathtt{W}}$. Restriction is similar to retrieval (Cockett, 1985).

Definition 5.4.2 *The intersection of two trees $\mathcal{T}_1$ and $\mathcal{T}_2$, which we denote $\mathcal{T}_1 \cap \mathcal{T}_2$, is the tree such that two states s_i and s_j map to the same leaf if and only if they map to the same leaves of both $\mathcal{T}_1$ and $\mathcal{T}_2$.*

The intersection of two trees $\mathcal{T}_1$ and $\mathcal{T}_2$ can be computed by appending the tree $\mathcal{T}_2 \mid \mathbf{c}_{\mathcal{L}}$ to each leaf $\mathcal{L}$ of $\mathcal{T}_1$. Let $\mathcal{T}_{\mathtt{U}}$ denote the tree that only distinguishes between values of state variable $S_{\mathtt{U}}$. Figure 5.2 shows $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}}$, $\mathcal{T}_{\mathtt{U}}$, as well as the intersection $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}} \cap \mathcal{T}_{\mathtt{U}}$. Intersection resembles to leaf composition (Moret, 1982) and logical operations on decision trees that represent indicator functions. Intersection is also similar to the merge operation of Boutilier et al. (2000).

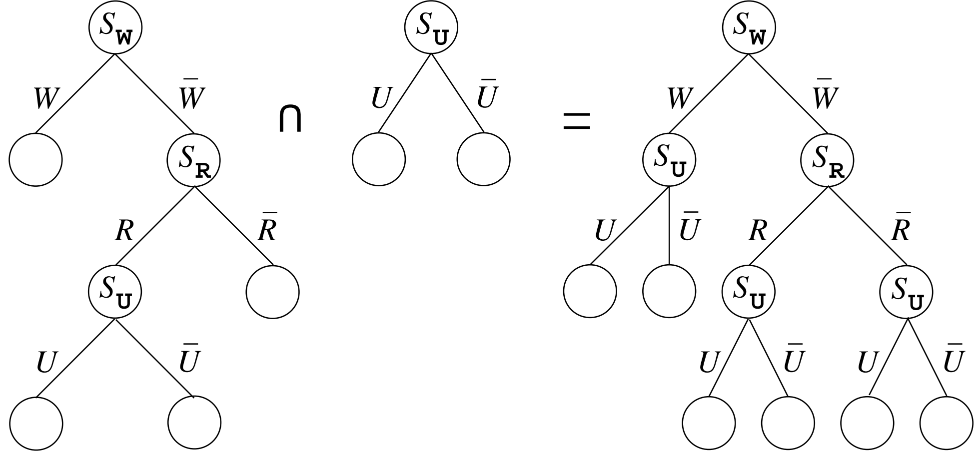

Figure 5.2. The trees $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}}$, $\mathcal{T}_{\mathtt{U}}$, and the intersection $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}} \cap \mathcal{T}_{\mathtt{U}}$

5.4.2 Constructing partitions for exit options

In this section, we show how to identify partitions that reduce the complexity of computing a multi-time model of the exit options discovered by the VISA algorithm. Let o be an exit option whose goal it is to reach the context $\mathbf{c}$ of the associated exit $\langle \mathbf{c}, a \rangle$. Recall that the VISA algorithm learns the policy π of an option in the form of a tree. Let $\mathcal{T}_\pi^o$ be the policy tree of exit option o. For example, Figure 5.3 shows the policy of the exit option associated with the exit $\langle (S_{\mathtt{L}} = L), \mathtt{BC} \rangle$ in the coffee task. Each leaf $\mathcal{L}$ of $\mathcal{T}_\pi^o$ is associated with a set of actions $A_{\mathcal{L}}$ selected by the policy π with non-zero probability in states that map to $\mathcal{L}$. The policy tree $\mathcal{T}_\pi^o$ induces a partition Λ_π of S such that two states $\mathbf{s}_i$ and $\mathbf{s}_j$ belong to the same block of Λ_π if and only if $\mathbf{s}_i$ and $\mathbf{s}_j$ map to the same leaf of $\mathcal{T}_\pi^o$. Λ_π is policy respecting since the policy is equal for all states that map to the same leaf of $\mathcal{T}_\pi^o$. Assume that one or several leaves of $\mathcal{T}_\pi^o$ correspond to the context $\mathbf{c}$ of the associated exit; then Λ_π is also termination respecting, since $\beta(\mathbf{s}_i) = 1$ for states that map to those leaves and $\beta(\mathbf{s}_i) = 0$ for all other states.

For each state variable S_d, we want to construct a partition Λ_d that has the stochastic substitution property, is policy respecting, termination respecting and re-

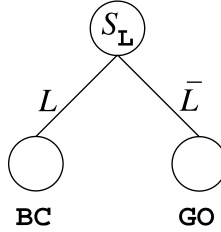

Figure 5.3. The policy of the exit option associated with the exit $\langle(S_{\mathtt{L}} = L), \mathtt{BC}\rangle$

spects state variable S_d. We begin by computing $\mathcal{T}_\pi^o \cap \mathcal{T}_d$, the intersection between $\mathcal{T}_\pi^o$ and the tree $\mathcal{T}_d$ that only distinguishes between values of $\mathbf{S}_d$ (cf. $\mathcal{T}_{\mathtt{U}}$ in Figure 5.2). The resulting tree induces a partition that is policy respecting and termination respecting because of $\mathcal{T}_\pi^o$, and respects state variable S_d because of $\mathcal{T}_d$.

To construct a partition Λ_d which also has the stochastic substitution property, we repeatedly apply MAKESSP in Algorithm 2 to $\mathcal{T}_\pi^o \cap \mathcal{T}_d$ until the structure of the tree does not change between successive iterations. The intuition is that in order for a tree to induce an SSP partition, the context $\mathbf{c}_{\mathcal{L}}$ at each leaf $\mathcal{L}$ of the tree has to be sufficient to determine possible transitions to other leaves of the tree as a result of executing an action $a \in A_{\mathcal{L}}$. If the context $\mathbf{c}_{\mathcal{L}}$ is insufficient to determine transitions, we refine the tree at $\mathcal{L}$. Each time the tree is refined, we label new leaves with the corresponding set of policy actions.

Algorithm 2 MAKESSP($\mathcal{T}$)

```
1: Input: Tree T
2: for each leaf L of T and each action a ∈ A_L
3:     N ← root of T
4:     REFINE(L, a, N)
```

As an example, consider the exit option o associated with the exit $\langle(S_{\mathtt{L}} = L), \mathtt{BC}\rangle$ in the coffee task. We want to identify $\Lambda_{\mathtt{W}}$, the partition associated with state variable $S_{\mathtt{W}}$. We start with the policy tree $\mathcal{T}_\pi^o$ of option o in Figure 5.3 and compute the intersection $\mathcal{T}_\pi^o \cap \mathcal{T}_{\mathtt{W}}$, shown in Figure 5.4. Next, we apply MAKESSP to $\mathcal{T}_\pi^o \cap \mathcal{T}_{\mathtt{W}}$. The context of the right-most leaf of $\mathcal{T}_\pi^o \cap \mathcal{T}_{\mathtt{W}}$ is $(S_{\mathtt{L}} = \overline{L}, S_{\mathtt{W}} = \overline{W})$. As a result of executing

Algorithm 3 REFINE($\mathcal{L}, a, \mathcal{N}$)

1: Input: Leaf $\mathcal{L}$, action a, tree node $\mathcal{N}$
2: **if** $\mathcal{N}$ is a leaf node, stop
3: $S_d \leftarrow$ state variable at node $\mathcal{N}$
4: determine the probability distribution $P_d(k \mid \mathbf{c}_{\mathcal{L}}, a)$ over possible values $k \in Val(S_d)$
5: **if** $\mathbf{c}_{\mathcal{L}}$ is insufficient to determine the probability distribution $P_d(k \mid \mathbf{c}_{\mathcal{L}}, a)$
6: append the tree $\mathcal{T}_d^a \mid \mathbf{c}_{\mathcal{L}}$ to leaf $\mathcal{L}$
7: **else for each** $k \in Val(S_d)$ such that $P_d(k \mid \mathbf{c}_{\mathcal{L}}, a) > 0$
8: $\mathcal{N}_k \leftarrow$ child k of node $\mathcal{N}$
9: REFINE($\mathcal{L}$, a, $\mathcal{N}_k$)

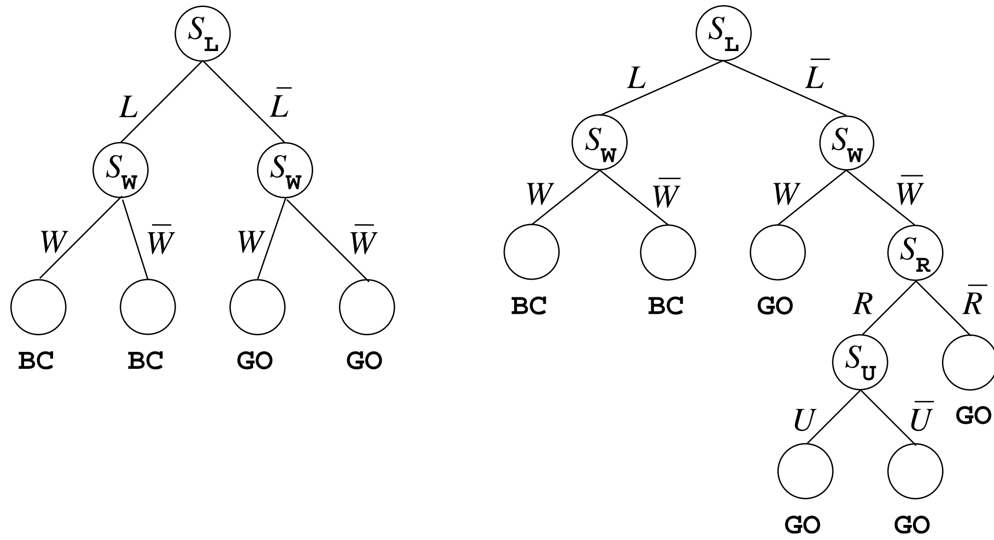

Figure 5.4. The tree $\mathcal{T}_\pi^o \cap \mathcal{T}_{\mathtt{W}}$ and the result of MAKESSP($\mathcal{T}_\pi^o \cap \mathcal{T}_{\mathtt{W}}$)

$\texttt{GO}$, this context is sufficient to determine a distribution over $Val(S_{\texttt{L}})$, the possible resulting values of $S_{\texttt{L}}$, the state variable at the root of the tree. However, to determine whether the robot will get wet, we also need to know if it is raining, and possibly whether the robot has an umbrella. Since the context $(S_{\texttt{L}} = \overline{L}, S_{\texttt{W}} = \overline{W})$ is insuffcient to determine a distribution over $Val(S_{\texttt{W}})$, we append $\mathcal{T}_{\texttt{W}}^{\texttt{GO}} \mid (S_{\texttt{L}} = \overline{L}, S_{\texttt{W}} = \overline{W})$ to the right-most leaf. The result is shown in Figure 5.4. Subsequent applications of MakeSSP does not change the structure of the tree, so the induced partition has all the desired properties.

The tree that results from repeatedly applying MakeSSP to $\mathcal{T}_{\pi}^{o} \cap \mathcal{T}_{d}$ induces a partition Λ_d that has the stochastic substitution property, is policy respecting, termination respecting, and respects state variable S_d. From Lemma 5.3.4 it follows that we can construct a reduced system of linear equations according to Equation (5.16) to compute the conditional probability model of state variable S_d for exit option o. The reduced system of linear equations has one row or column for each block of Λ_d instead of one row or column for each state in S. In our example, the resulting partition $\Lambda_{\texttt{W}}$ has 6 blocks as compared to the 64 states in S.

To construct a partition Λ_R for computing the expected reward of exit option o, we proceed in a similar way. Start with the policy tree $\mathcal{T}_{\pi}^{o}$. At each leaf $\mathcal{L}$ of $\mathcal{T}_{\pi}^{o}$, append the tree $\cap_{a \in A_{\mathcal{L}}} \mathcal{T}_{R}^{a} \mid \mathbf{c}_{\mathcal{L}}$, where $\mathcal{T}_{R}^{a}$ is the tree that determines the expected reward associated with action a in the DBN model. The partition induced by the resulting tree is reward respecting, policy respecting, and termination respecting. Then apply MakeSSP repeatedly until convergence in order to construct a partition which also has the stochastic substitution property. From Lemma 5.3.5 it follows that we can construct a reduced system of linear equations according to Equation (5.17) to compute the expected reward associated with exit option o.

5.4.3 Distribution irrelevance

Dietterich (2000b) defined a condition that he calls result distribution irrelevance: a subset of the state variables may be irrelevant for the resulting distribution of an activity. This condition only exists in the undiscounted case, i.e., for $\gamma = 1$. Otherwise, the time it takes the activity to terminate influences subsequent reward.

We can take advantage of distribution irrelevance to compute the multi-time model of an exit option when $\gamma = 1$. Let o be the exit option associated with the exit $\langle \mathbf{c}, a \rangle$. Since o terminates in the context $\mathbf{c}$, we know the value of each state variable in the set $\mathbf{C} \subseteq \mathbf{S}$ right before action a is executed. In other words, the values of state variables in the set $\mathbf{C}$ prior to executing o are irrelevant for the resulting distribution of o.

Because of distribution irrelevance, we do not need to solve Equation (5.16) for state variables in the set $\mathbf{C}$. Instead, the conditional probabilities associated with state variable $S_d \in \mathbf{C}$ and option o are given by the tree $\mathcal{T}_d^a \mid \mathbf{c}$. For example, as a result of executing the exit option associated with the exit $\langle (S_{\mathtt{L}} = L), \mathtt{BC} \rangle$ in the coffee task, the value of state variable $S_{\mathtt{L}}$ is L right before executing $\mathtt{BC}$. The conditional probabilities of $S_{\mathtt{L}}$ are given by $\mathcal{T}_{\mathtt{L}}^{\mathtt{BC}} \mid (S_{\mathtt{L}} = L)$, which is a tree with just a root node. As a result of executing the option that acquires coffee, the location of the robot is always the coffee shop, regardless of its previous location.

We can also simplify computation of conditional probabilities for state variables that are unaffected by actions that the policy selects. Let $\mathbf{U}^o \subseteq \mathbf{S}$ denote the subset of state variables whose value does not change as a result of executing any action selected by the policy π of exit option o. For the exit option o associated with exit $\langle (S_{\mathtt{L}} = L), \mathtt{BC} \rangle$ in the coffee task, $\mathbf{U}^o = \{S_{\mathtt{U}}, S_{\mathtt{R}}, S_{\mathtt{C}}, S_{\mathtt{H}}\}$, since the values of these state variables do not change as a result of executing $\mathtt{GO}$, the only action selected by the policy of o. The conditional probabilities of state variables in the set $\mathbf{U}^o$ unaffected by the policy actions are also given by $\mathcal{T}_d^a \mid \mathbf{c}$.

5.4.4 Summary of the algorithm

At this point, we stop to summarize the steps of our algorithm. Algorithm 4 outlines our algorithm for computing a compact multi-time model of an exit option.

Algorithm 4 Computing a compact multi-time option model

1: Input: DBN model of a factored MDP $\mathcal{M}$, exit option o
2: let $\langle \mathbf{c}, a \rangle$ be the exit associated with o, where $\mathbf{c}$ is an assignment to $\mathbf{C} \subseteq \mathbf{S}$
3: let O_o be the set of options in the option SMDP $\mathcal{M}_o$ associated with o
4: let $\mathbf{U}^o \subseteq \mathbf{S}$ be the set of state variables unaffected by options in the set O_o
5: let $\mathcal{T}_\pi^o$ be the tree storing option o's policy π
6: **for each** state variable $S_d \in \mathbf{S}$
7:     **if** $S_d \in \mathbf{C} \cup \mathbf{U}^o$
8:         $\mathcal{T}_d^o \leftarrow \mathcal{T}_d^a \mid \mathbf{c}$
9:         copy labels at the leaves of $\mathcal{T}_d^a \mid \mathbf{c}$ to obtain the probability distribution P_d^o
10:     **else**
11:         $\mathcal{T}_d^o \leftarrow$ MakeSSP$(\mathcal{T}_\pi^o \cap \mathcal{T}_d)$
12:         let Λ_d be the partition of S induced by $\mathcal{T}_d^o$
13:         set up and solve Equation (5.16) to obtain the probability distribution P_d^o
14: Let $A_\mathcal{L} \subseteq A$ be the set of actions selected by the policy π at leaf $\mathcal{L}$ of $\mathcal{T}_\pi^o$
15: Let $\mathcal{T}_R$ be the tree that results from appending $\cap_{a' \in A_\mathcal{L}} \mathcal{T}_R^{a'} \mid \mathbf{c}_\mathcal{L}$ at each leaf $\mathcal{L}$ of $\mathcal{T}_\pi^o$
16: $\mathcal{T}_R^o \leftarrow$ MakeSSP$(\mathcal{T}_R)$
17: let Λ_R be the partition of S induced by $\mathcal{T}_R^o$
18: set up and solve Equation (5.17) to obtain the expected reward distribution R^o

5.5 DBN model for options

When we have computed the conditional probability tree associated with each state variable for an exit option o, as well as a tree representing expected reward, we can construct a DBN for the option just like for primitive actions. Figure 5.5 shows the DBN for the exit option associated with the exit $\langle (S_\mathtt{L} = L), \mathtt{BC} \rangle$ in the coffee task when $\gamma = 1$ and we take advantage of distribution irrelevance. Note that there is no edge to state variable $S_\mathtt{L}$, which indicates that the resulting location does not depend on any of the state variables.

The conditional probability tree associated with state variable $S_\mathtt{W}$ has 6 leaves, and the tree associated with expected reward has 12 leaves. The complexity of inverting a general $n \times n$ matrix is $O(n^3)$, i.e., the time it takes to solve each equation of

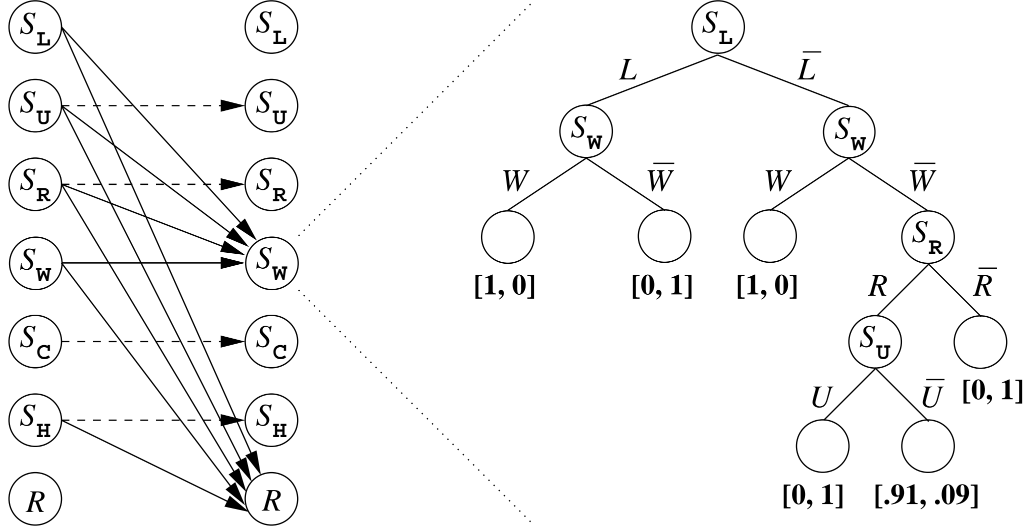

Figure 5.5. DBN for the option associated with $\langle (S_{\mathtt{L}} = L), \mathtt{BC} \rangle$

Table 5.1. Complexity of computing a multi-time model for each exit option

Exit	Change	Computation time
$\langle (S_{\mathtt{L}} = L), \mathtt{BC} \rangle$	$\overline{C} \to C$	$1,700k$
$\langle (S_{\mathtt{L}} = \overline{L}), \mathtt{DC} \rangle$	$C \to \overline{C}$	$1,700k$
$\langle (S_{\mathtt{L}} = \overline{L}, S_{\mathtt{C}} = C), \mathtt{DC} \rangle$	$\overline{H} \to H$	$22,000k$
$\langle (S_{\mathtt{L}} = \overline{L}), \mathtt{GU} \rangle$	$\overline{U} \to U$	$1,700k$
$\langle (S_{\mathtt{U}} = \overline{U}, S_{\mathtt{R}} = R), \mathtt{GO} \rangle$	$\overline{W} \to W$	0

the multi-time model is approximately kn^3, where k is a constant. Partitions and distribution irrelevance reduce the total computation time from $64^3k = 260,000k$ in the exact case to $6^3k + 12^3k = 1,700k$. Table 5.1 shows the total time it takes to compute a compact model of the exit option associated with each exit in the coffee task. The complexity of computing a compact model of each exit option is vastly reduced using our technique. Naturally, the efficiency of our technique depends on the nature of the task; the technique will work better if there are at least two strongly connected components in the causal graph and if there is a limited number of exits.

Since the DBN model of an option is in the same form as the DBN models of primitive actions, we can treat the option as a single unit and apply any of the algorithms that take advantage of compact representations. In addition, the DBN model makes it possible to apply our technique to nested options, i.e., options selecting between other options. Once the policy of an option has been learned, we can construct its DBN model and use that model both to learn the policy of a higher-level option and later to construct a DBN model of the higher-level option.

5.6 Experimental results

We used our approach to construct DBN models of options discovered by the VISA algorithm in the Taxi task (Dietterich, 2000a). We used Structured Policy Iteration, or SPI (Boutilier et al., 1995), which takes advantage of DBN models, to compute a policy of each option. At the top level, we ran SPI to compute a policy that selects between the identified options. Note that SPI at the top level is only possible after we construct the multi-time model of each exit option. We compared the performance of this scheme, which we call hierarchical SPI, with SPI on the flat task. Figure 5.6 illustrates the average reward over 100 trials for the two schemes. The graph for hierarchical SPI includes the time it takes VISA to decompose the task, running SPI on each of four individual options, applying our technique for constructing a DBN model of each option, and running SPI at the top level. Even so, hierarchical SPI still converges much faster than flat SPI. Since the VISA algorithm treats options as stand-alone tasks, it can only achieve recursive optimality at best, as opposed to hierarchical optimality (Dietterich, 2000a). We believe this accounts for the slightly lower level of reward of hierarchical SPI after convergence.

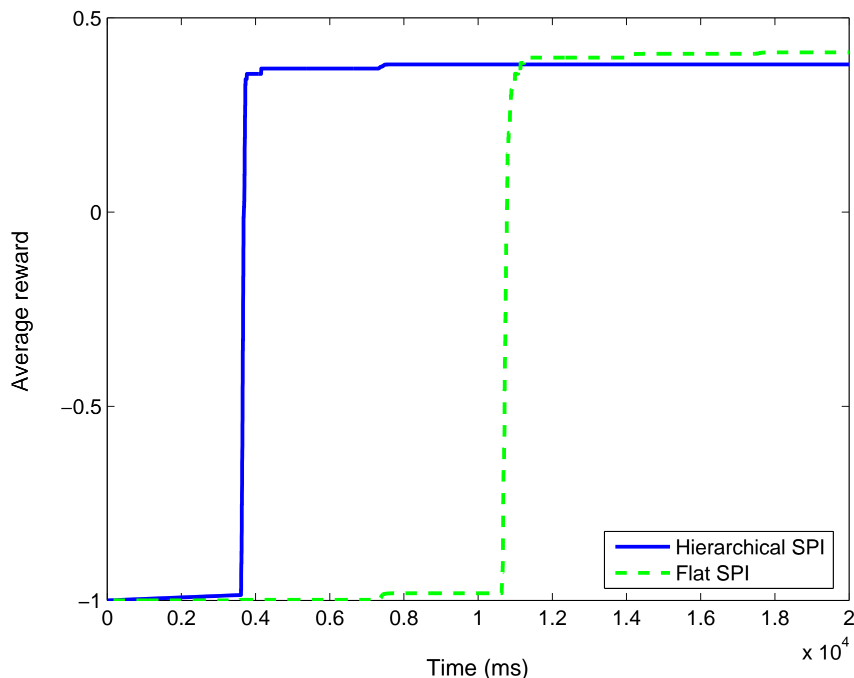

Figure 5.6. Hierarchical vs. flat SPI in the Taxi task

5.7 Discussion

Our algorithm for constructing DBN models for options relies on partitions with a preestablished set of properties to reduce the complexity of computing compact option models. The requirement that the partitions should have all of these properties is quite strong. A possible line of future research is to relax or approximate the required properties of partitions, which could lead to even more efficient computation of option models, albeit with some loss of accuracy. An analysis of the resulting approximation could help determine a tradeoff between the complexity of computing compact option models and the accuracy of the resulting model.

We also made a strong independence assumption in order to reduce the complexity of computing a compact option model. Our algorithm assumes that the value of a state variable that results from executing an option is independent of the resulting values of other state variables. Since an option takes variable time to execute, the

option passes through many states during execution. The independence assumption only holds if the resulting values of state variables are independent regardless of which state the option is currently in. In many cases, our independence assumption induces an approximation error. If possible, we would like to establish bounds on this approximation error to analyze the accuracy of our algorithm.

5.8 Related work

Sutton et al. (1999) developed the multi-time model of options that we used in this chapter. The multi-time model includes an estimate of the transition probabilities and expected reward of options. Using the multi-time model of an option, it is possible to treat the option as a single unit during learning and planning. SMDP value learning (Sutton et al., 1999) uses the multi-time model to learn values or action-values in an SMDP. SMDP planning (Sutton et al., 1999) uses the multi-time model to perform planning in an SMDP, similar to policy iteration.

There has not been a lot of other research on constructing models of activities. The completion function in MAXQ (Dietterich, 2000a) can be viewed as a model of the expected reward of the activity. The H-Tree algorithm (Jonsson and Barto, 2001) that we presented in Chapter 3 estimates a model of the transition probabilities and expected reward of the options in a partially observable semi-Markov decision process, or POSMDP. This model is compact since the history of an option is partitioned using a U-Tree, but is usually only approximate.

CHAPTER 6

LEARNING DBN MODELS OF FACTORED MDPS

Recall that the DBN model of a factored MDP compactly represents the transition probabilities and expected reward of the MDP. In a DBN model, actions are defined in terms of probabilistic causes and effects on the state variables of the MDP. In previous chapters, we assumed that we had access to a DBN model prior to learning in a factored MDP. We developed several algorithms that exploit DBN models to decompose and simplify a task. In addition, there exist several other algorithms that use DBN models to efficiently solve factored MDPs.

A DBN model may not be available prior to solving a factored MDP. In this chapter, we address the problem of learning DBN models from experience. There exist algorithms in the literature for learning the structure of Bayesian networks (Buntine, 1991; Friedman et al., 1998; Heckerman et al., 1995). However, these algorithms assume that a data set is given, whereas solution techniques for MDPs typically have to gather data in the form of transitions and reward through interaction with the environment. The complexity of learning DBNs heavily depends on the time it takes to collect data. It is possible to accelerate data collection by selecting high-quality data instances through a process called active learning. Researchers have developed techniques for active learning of Bayesian networks (Murphy, 2001; Steck and Jaakkola, 2002; Tong and Koller, 2001). These techniques perform experiments by clamping a subset of the variables to fixed values and sampling over the remaining variables.

We assume that it is only possible to sample MDPs along trajectories, not in arbitrary states. The only way to gather information about transitions and reward is by executing an action in the current state. Since it is not possible to simulate the effect of actions in hypothetical states, we cannot perform experiments by clamping a subset of the variables to fixed values. Consequently, we cannot apply existing techniques for active learning. However, there is still an opportunity to perform active learning of DBNs in factored MDPs. Because the DBN model of a factored MDP consists of one DBN for each action, by selecting an action we effectively select a DBN for which to collect data. It is thus possible to consider policies for action selection whose utility lie in efficient data collection.

We develop an algorithm for learning DBNs that grows trees representing the conditional probabilities of the DBNs. Our algorithm collects data instances by executing actions and extends a tree as soon as a minimum number of data instances correspond to each relevant value of each split variable. The algorithm uses the Bayesian Information Criterion, or BIC (Schwartz, 1978), and the likelihood-equivalent Bayesian Dirichlet metric, or BDe (Heckerman et al., 1995), to evaluate potential refinements. We assume that no data is available to begin with and develop a method for active learning of DBNs to accelerate data collection. The time to collect data is minimized if the distribution of data instances across values of each potential split variable is perfectly uniform. We use the entropy of the distributions to measure uniformity and select actions that maximize the total entropy of the distributions.

In some tasks, the BIC and BDe scores fail to detect most of the refinements necessary to learn an accurate DBN model. This is a serious issue since algorithms that take advantage of DBNs to solve factored MDPs depend on the accuracy of the DBN model. We address this issue by applying regularization (Poggio and Girosi, 1990) to the BIC score. The BIC score is composed of a log likelihood term and a penalty term. This quantity fits nicely into the regularization framework if we

multiply the penalty term by a parameter λ. Empirical results show that varying λ can increase the accuracy of the learned DBN model.

Our work is related to the problem of exploration in reinforcement learning (Sutton and Barto, 1998). Existing exploration techniques do not learn DBN models of MDPs. Since there exist several efficient algorithms that use DBNs to solve factored MDPs, there is a benefit to learning this representation. Our approach does not require enumeration of the state space, as opposed to several exploration methods. Since we want to scale to large state spaces, we do not want to store quantities whose size is proportional to the number of states.

6.1 Learning the structure of Bayesian networks

Structure learning is the problem of finding the Bayesian network $B = \langle G, \theta \rangle$ that best fits a data set $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$. Each element $\mathbf{x} \in D$ is an assignment of values to the set of variables $\mathbf{X}$ of the Bayesian network. A common approach is to compute the posterior probability distribution $P(B \mid D)$ over Bayesian networks and choose the network that maximizes $P(B \mid D)$. Two common approximations of $P(B \mid D)$ are the Bayesian Information Criterion, or BIC (Schwartz, 1978), and the likelihood-equivalent Bayesian Dirichlet metric, or BDe (Heckerman et al., 1995). From Bayes theorem it follows that $P(B \mid D) \propto P(D \mid B)P(B)$. The BIC score makes the approximation

$$\log[P(D \mid B)P(B)] \approx L(D \mid B) - \frac{|\theta|}{2} \log |D|, \tag{6.1}$$

where $L(D \mid B)$ is the log likelihood of D given B. If the data set D contains no missing values, the log likelihood decomposes as

$$L(D \mid B) = \sum_i \sum_j \sum_k N_{ijk} \log \theta_{ijk},$$

where N_{ijk} is defined as the number of data points $\mathbf{x} \in D$ such that $f_{\mathbf{Pa}(X_i)}(\mathbf{x}) = j$ and $f_{\{X_i\}}(\mathbf{x}) = k$, and $\theta_{ijk} = P(X_i = k \mid \mathbf{Pa}(X_i) = j)$. The log likelihood is maximized for $\theta_{ijk} = N_{ijk} / \sum_k N_{ijk}$. The BDe score makes the approximation

$$P(D \mid B)P(B) \approx \prod_i \prod_j \frac{\Gamma(\sum_k N'_{ijk})}{\Gamma(\sum_k [N'_{ijk} + N_{ijk}])} \prod_k \frac{\Gamma(N'_{ijk} + N_{ijk})}{\Gamma(N'_{ijk})}, \tag{6.2}$$

where N'_{ijk} are hyperparameters of a Dirichlet prior and $\Gamma(x)$ is the Gamma function.

Finding the Bayesian network with highest BIC or BDe score is NP-complete (Chickering et al., 1995). However, both scores decompose into a sum of terms for each variable X_i and each value j of $\mathbf{Pa}(X_i)$ and k of X_i (in the case of the BDe score, we need to take the logarithm first). The score only changes locally when we add or remove edges between variables in the directed acyclic graph G of the network. Researchers have developed hill-climbing algorithms that perform greedy search to find high-scoring Bayesian networks by repeatedly adding or removing edges between variables in G (Buntine, 1991; Heckerman et al., 1995). These algorithms have been extended to DBNs (Friedman et al., 1998).

6.2 Learning a DBN model of factored MDPs

We develop an algorithm for learning DBN models of factored MDPs through interaction with the environment. Our algorithm builds a tree $\mathcal{T}_i^a$ for each pair of a state variable S_i and action a, approximating the conditional probabilities of S_i as a result of executing a. The family of trees for a implicitly defines the DBN for a. There is an edge between state variables S_j and S_i in the DBN if at least one node in $\mathcal{T}_i^a$ distinguishes between values of S_j.

To build the tree $\mathcal{T}_i^a$, the algorithm starts with a small tree and collects data by executing actions in the environment. Each time action a is executed, the algorithm records a data instance $\mathbf{x}^a$. In a DBN, a data instance consists of two assignments

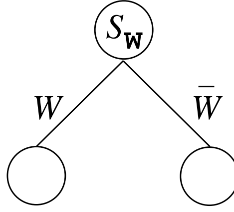

Figure 6.1. Intermediate configuration of the tree $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}}$ during learning

to each state variable: one assignment corresponding to the current time step, and one assignment corresponding to the next time step. Since the DBN model has one node for each state variable and one node for expected reward, a data instance $\mathbf{x}^a = (\mathbf{s}, r, \mathbf{s}')$ consists of the state $\mathbf{s}$ prior to executing a, the reward r received as a result, and the state $\mathbf{s}'$ following execution of a. The DBN model assumes that the reward received during the previous time step does not influence the resulting values of state variables or the expected reward, so a data instance does not need to include the previous reward.

The tree $\mathcal{T}_i^a$ stores the probability distribution over values of state variable S_i as a result of executing a, conditional on the previous values of the state variables. In other words, the tree $\mathcal{T}_i^a$ makes distinctions between values of state variables prior to executing a. It follows that each data instance $\mathbf{x}^a$ maps to exactly one leaf of $\mathcal{T}_i^a$ according to the state $\mathbf{s}$ that was recorded prior to executing a. A data instance $\mathbf{x}^a$ is stored at the unique leaf that it maps to. We say that a leaf is empty if its corresponding set of data instances is empty.

For example, say that we are growing the tree $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}}$ storing the conditional probabilities of state variable $S_{\mathtt{W}}$ as a result of executing action `GO` in the coffee task. Assume that the current configuration of $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}}$ is the one in Figure 6.1. Also assume that we execute action `GO` in state $\mathbf{s} = (L, \overline{U}, R, \overline{W}, C, \overline{H})$, and that as a result, the process transitions to state $\mathbf{s}' = (\overline{L}, \overline{U}, R, W, C, \overline{H})$ and the learning agent receives a

reward $r = 0.1$. Our algorithm records a data instance $\mathbf{x}^{\texttt{GO}} = (\mathbf{s}, r, \mathbf{s}')$, which maps to the right-most leaf of $\mathcal{T}_{\texttt{W}}^{\texttt{GO}}$, since state $\mathbf{s}$ assigns the value $\overline{W}$ to state variable $S_{\texttt{W}}$.

A refinement at a leaf distinguishes between values of a state variable S_j prior to executing action a and introduces a new leaf of $\mathcal{T}_i^a$ for each value of S_j. A state variable S_j is only considered for refinement if no internal nodes on the path from the root to the leaf of $\mathcal{T}_i^a$ already distinguish between values of S_j. The BIC and BDe scores decompose into a local score for each leaf of $\mathcal{T}_i^a$. Our algorithm evaluates a refinement by comparing the total score of the new leaves with the score of the old leaf. If at least one refinement increases the overall score, the algorithm retains the refinement that results in the largest increase. Regardless of the outcome, data instances at the old leaf are discarded. Our approach is more sophisticated than adding edges in the graph of the DBN, since trees store conditional probabilities more compactly than tables.

When the algorithm evaluates a refinement, it performs a statistical test to compare the posterior probabilities of two Bayesian networks given the data. It is well known that the accuracy of statistical tests, such as Chi-square, depends on having enough examples in each bin. At each leaf, and for each potential split variable S_j, the algorithm maintains a distribution vector M. Each entry M_k of the vector indicates the number of data instances at the leaf that assign the value k to S_j. When the algorithm evaluates a refinement over S_j, the distribution vector M determines how the data instances at the leaf will be distributed to the new leaves of $\mathcal{T}_i^a$. We define a parameter K and let our algorithm evaluate a refinement as soon as at least K data instances map to each non-empty leaf for each split variable.

In our example, five state variables are potential split variables at the right-most leaf of the tree $\mathcal{T}_{\texttt{W}}^{\texttt{GO}}$: $S_{\texttt{L}}$, $S_{\texttt{U}}$, $S_{\texttt{R}}$, $S_{\texttt{C}}$, and $S_{\texttt{H}}$. No internal node on the path from the root of the tree to the right-most leaf makes a distinction between values of any of these state variables. For each potential split variable, our algorithm maintains

a distribution vector M. The data instance $\mathbf{x}^{\texttt{GO}} = (\mathbf{s}, r, \mathbf{s}')$ that was recorded in our example contributes a count to the value L in the distribution vector M corresponding to potential split variable $S_{\texttt{L}}$, since state $\mathbf{s}$ of the data instance assigns the value L to $S_{\texttt{L}}$. Similarly, the data instance $\mathbf{x}^{\texttt{GO}}$ contributes a count to the distribution vector of each other potential split variable.

In some tasks, the BIC and BDe scores fail to detect most of the refinements necessary to learn an accurate DBN model. The BIC score in Equation (6.1) is composed of a log likelihood term, which measures the likelihood of the data given a network, and a penalty term, which penalizes a network for having many parameters. The penalty term causes the BIC score to be less sensitive to improvements to the log likelihood since each refinement increases the number of parameters. We use regularization (Poggio and Girosi, 1990) to address this issue. In regularization, a functional is defined as the sum of a fidelity term and a stabilizer term. The stabilizer term is weighted by a parameter λ. We can multiply the penalty term of the BIC score by a parameter λ to put it in the form of a fidelity term and a stabilizer term:

$$\log[P(D \mid B)P(B)] \approx L(D \mid B) - \lambda \frac{|\theta|}{2} \log |D|, \tag{6.3}$$

such that λ controls the magnitude of the penalty for having many parameters.

6.2.1 Active learning

Efficient data collection should gather sufficient data as quickly as possible. Since our algorithm requires at least K data instances to map to each non-empty leaf, the distribution of data instances across new leaves should be as uniform as possible for each possible refinement. The more skewed the distribution is after collecting enough data instances, the longer it takes to collect sufficient data to evaluate refinements. We devise the following scheme for active learning of DBNs. Recall that the state $\mathbf{s}$ prior to executing action a determines which leaf of $\mathcal{T}_i^a$ a data instance $\mathbf{x}^a$ maps

to. In other words, we already know which leaf a data instance will map to prior to executing a. For each state variable S_i and each action a, we find the leaf of the tree $\mathcal{T}_i^a$ that the next data instance will map to. When deciding which action to execute, we look at how the distribution vectors at each of those leaves would change as a result of executing the corresponding action. To evaluate the change, we compute the entropy $H(M)$ of each distribution vector M:

$$H(M) = -\sum_k \theta_k \log \theta_k,$$

where $\theta_k = M_k / \sum_j M_j$. $H(M)$ is a non-negative function which is maximized when all entries of M are equal. An increase in $H(M)$ means that the distribution is becoming more uniform; a decrease means that it is becoming more skewed. The change in $H(M)$ can be computed in constant time. In each state, the active learning scheme selects the action with largest total increase in the value of $H(M)$. With probability $\epsilon \in [0, 1]$, or if no action results in an increase of $H(M)$, the scheme selects a random action. Our active learning scheme only implicitly changes the frequency with which leaves are visited; it assumes that each leaf is visited relatively frequently.

For example, assume that we are learning the DBN model of the coffee task, and that the current state is $\mathbf{s} = (L, \overline{U}, R, \overline{W}, C, \overline{H})$. Further assume that the current configuration of the tree $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}}$ is the one in Figure 6.1, and that the distribution vector associated with state variable $S_{\mathtt{L}}$ is $M = [5, 8]$ at the right-most leaf. In other words, there are 5 data instances stored at the leaf that assign the value L to $S_{\mathtt{L}}$, and 8 data instances that assign the value $\overline{L}$ to $S_{\mathtt{L}}$. If we were to execute action GO in the current state, the algorithm would record a new data instance $\mathbf{x}^{\mathtt{GO}} = (\mathbf{s}, r, \mathbf{s}')$. The algorithm does not yet know what r and $\mathbf{s}'$ will be. However, the algorithm does know that since the current state $\mathbf{s}$ assigns the value $\overline{W}$ to state variable $S_{\mathtt{W}}$, the new data instance $\mathbf{x}^{\mathtt{GO}}$ will map to the right-most leaf of $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}}$.

We also show how the algorithm computes the change in entropy. We can write the entropy $H(M)$ in the following way, letting $N = \sum_j M_j$:

$$\begin{aligned} H(M) &= -\sum_k \frac{M_k}{N} \log \frac{M_k}{N} = \\ &= -\sum_k \frac{M_k}{N} (\log M_k - \log N) = \\ &= -\frac{1}{N} \sum_k M_k \log M_k + \frac{\log N}{N} \sum_k M_k = \\ &= -\frac{1}{N} \sum_k M_k \log M_k + \log N. \end{aligned}$$

Let M' be the distribution vector after adding a count to the entry M_i of M. It follows that $M'_i = M_i + 1$, that $M'_k = M_k$ for each $k \neq i$, and that $N' = \sum_j M'_j = N + 1$. Then we can write $H(M')$ in the following way:

$$\begin{aligned} H(M') &= -\frac{1}{N'} \sum_k M'_k \log M'_k + \log N' = \\ &= -\frac{1}{N+1} \sum_{k \neq i} M_k \log M_k - \frac{1}{N+1} (M_i + 1) \log(M_i + 1) + \log(N+1) = \\ &= \frac{N}{N+1} \left[H(M) + \frac{M_i \log M_i}{N} - \log N \right] - \frac{(M_i + 1) \log(M_i + 1)}{N+1} + \log(N+1). \end{aligned}$$

In our example, the entropy prior to recording a new data instance is

$$H(M) = -\frac{1}{5+8} (5 \log 5 + 8 \log 8) + \log(5+8) \approx 0.666.$$

The current state $\mathbf{s}$ assigns the value L to state variable $S_{\mathtt{L}}$, so the resulting distribution vector is $M' = [6, 8]$. The change in entropy as a result of executing `GO` is

$$H(M') - H(M) \approx \frac{13}{14} \left[0.666 + \frac{5 \log 5}{13} - \log 13 \right] - \frac{6 \log 6}{14} + \log(14) - 0.666 \approx 0.017.$$

To evaluate action GO, the algorithm has to compute the change in entropy for each potential split variable at the right-most leaf of $\mathcal{T}_{\mathtt{W}}^{\mathtt{GO}}$. In addition, the algorithm has to evaluate the change in entropy for trees that store the conditional probabilities of other state variables as a result of executing GO, following the same strategy.

6.2.2 Summary of the algorithm

At this point, we stop to summarize the steps of our algorithm. Algorithm 5 outlines our algorithm for active learning of a DBN model of a factored MDP.

Algorithm 5 Active learning of a DBN model

```
 1: Input: Factored MDP M, parameters K, ϵ
 2: initialize trees T_i^a and distributions M
 3: do until convergence
 4:     for each action, compute the total entropy of distributions at corresponding leaves
 5:     with probability ϵ, or if the total entropy of each action is less than 0
 6:         a ← random action
 7:     else
 8:         a ← action with highest total entropy
 9:     execute action a and record a data instance x^a = (s, r, s')
10:     for each state variable S_i ∈ S
11:         append x^a to the corresponding leaf of tree T_i^a
12:         update the distributions M
13:         if at least K data instances map to each non-empty leaf
14:             for each potential split variable S_j
15:                 evaluate the local BIC or BDe score of the refinement
16:             if at least one refinement increases the BIC or BDe score
17:                 retain the refinement that increases the score the most
18:                 redistribute the data instances at the leaf to the new leaves
```

6.3 Results

We ran experiments with our DBN learning approach in the coffee task (Boutilier et al., 1995), the Taxi task (Dietterich, 2000a), and the same simplified autonomous guided vehicle (AGV) task that we used in Chapter 4 (Ghavamzadeh and Mahadevan, 2001). In each task, we compared our active learning scheme with passive learning, i.e., random action selection. We had access to the true DBN model of each task and

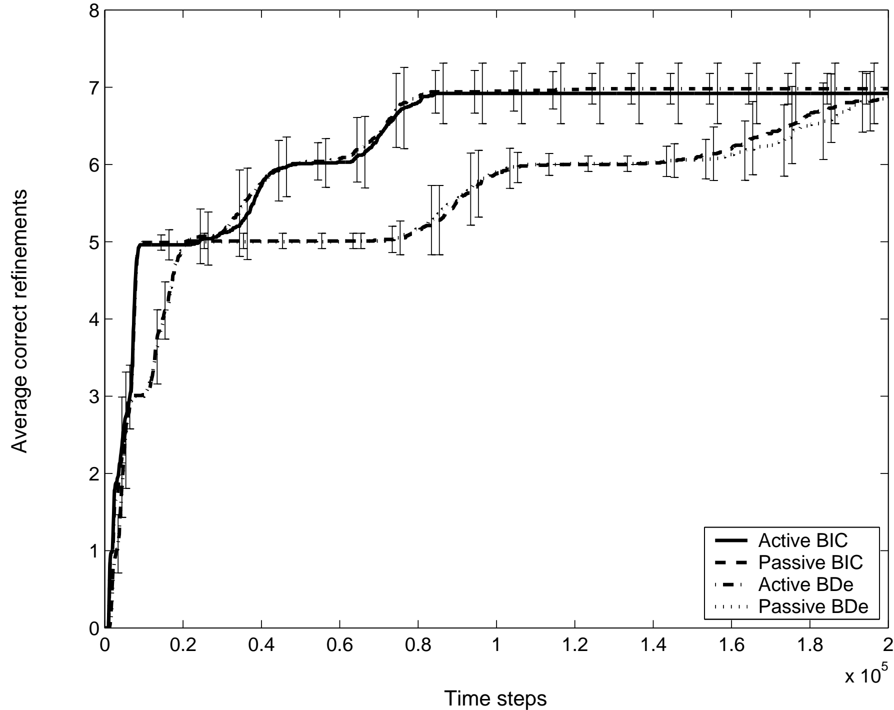

Figure 6.2. Results of learning DBNs in the coffee task

knew how many refinements of the conditional probability trees were necessary to learn the true model. Figure 6.2 shows results of our experiments in the coffee task. The graph shows the number of correct refinements (out of 7) detected over time, averaged over 100 trials. The error bars show the standard deviation of the number of correct refinements across trials. For each tree $\mathcal{T}_i^a$, we used the parameter values $\epsilon = 0.3$, $K = 50N_i$, where N_i is the number of values of the state variable S_i whose conditional probabilities $\mathcal{T}_i^a$ approximates. Note that active learning outperformed passive learning and that the BIC and BDe scores performed almost identically.

In the Taxi task, the BIC and BDe scores fail to detect most of the refinements necessary to learn the true DBN model. We tested our modification to the BIC score in Equation (6.3) to see if regularization can improve the accuracy of the learned DBN model. Figure 6.3 shows results of the experiments in the Taxi task, averaged over 25 trials. In the Taxi task, the true DBN model requires 21 refinements. We

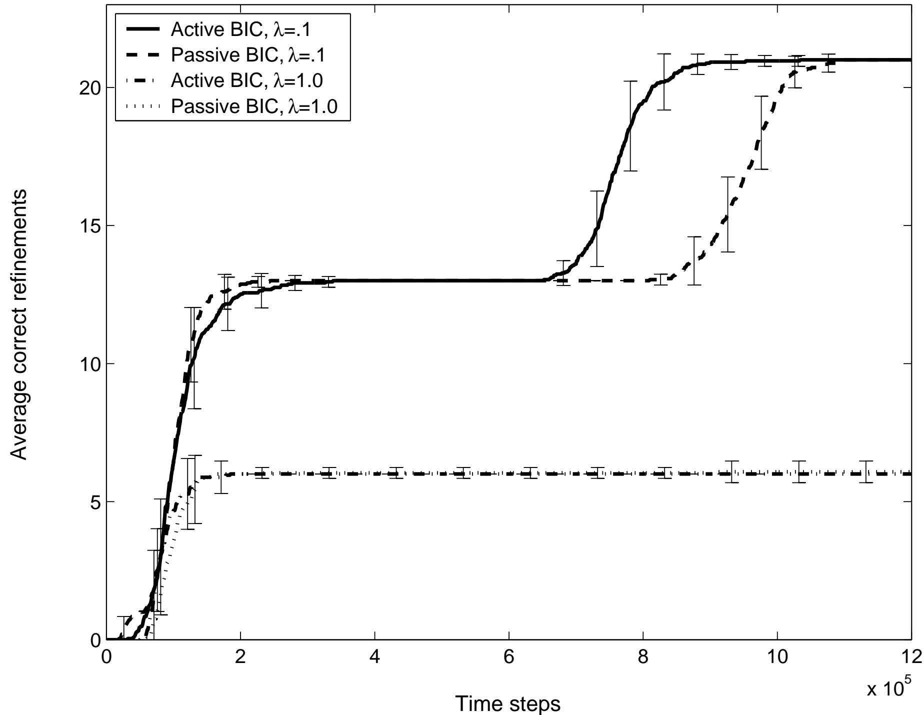

Figure 6.3. Results of learning DBNs in the Taxi task

used $\epsilon = 0.6$, $K = 50N_i$, and report results of the BIC score for $\lambda = 0.1$ and $\lambda = 1$. The BDe score performed identically to the BIC score for $\lambda = 1$. Note that active and passive learning using the original BIC score ($\lambda = 1$) failed to detect many of the refinements of the true DBN model. With $\lambda = 0.1$, active and passive learning detected all of the refinements.

We simplified the AGV task by reducing the number of machines to 2 and made it fully observable by setting the processing time of machines to 0. The resulting task has 75,000 states and 6 actions, and the true DBN model requires 162 refinements. Figure 6.4 shows results of the experiments in the AGV task, averaged over 5 trials. We used $\epsilon = 0.6$, $K = 50N_i$, and report results of the BIC score for $\lambda = 0.1$ and $\lambda = 1$. There are several interesting things to notice. First, learning was very slow. We collected data for 200,000,000 time steps, and it is not clear that the algorithms even converged. The learned DBN model did not come close to the true model, even

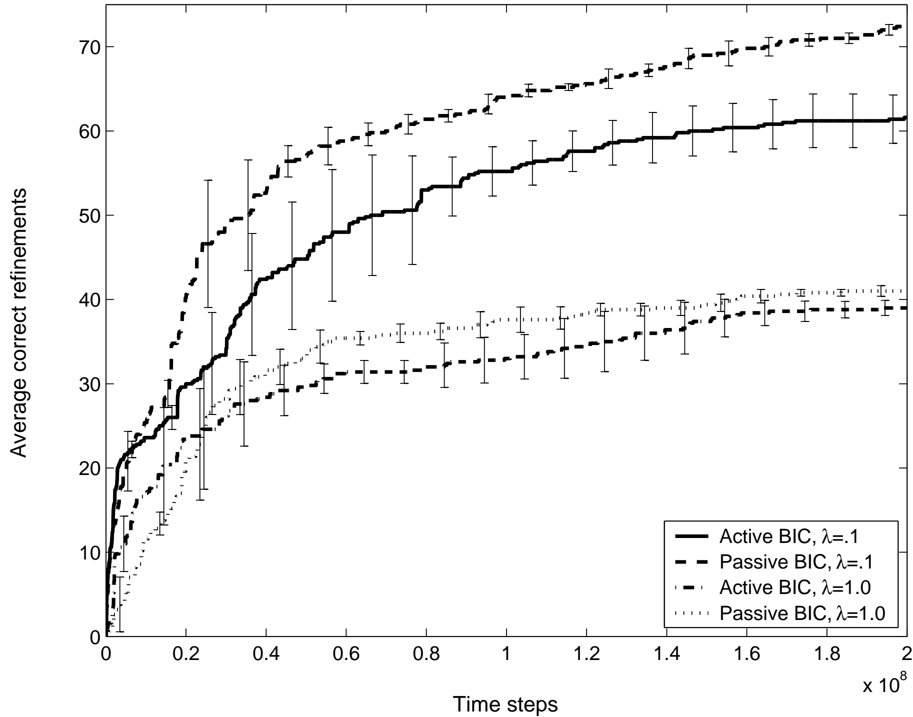

Figure 6.4. Results of learning DBNs in the AGV task

for $\lambda = 0.1$. Also, passive learning actually outperformed active learning in the AGV task. We believe this is due to the fact that our active learning scheme selects actions based on local information, which we elaborate on in the conclusion. The results of the experiments in the AGV task indicate that learning DBN models of factored MDPs is a challenging problem, even using state-of-the-art metrics such as the BIC and BDe scores.

6.4 Discussion

Our algorithm selects actions according to an active learning scheme based on local information, i.e., how the distributions change locally as a result of executing actions. This works well in tasks with limited size when all states are visited relatively frequently. However, as experimental results show, in large tasks our scheme may fail to explore large regions of the state space, prefering to maintain uniformity in the

current region. To ensure that most or all of the state space is visited it is necessary to select actions based on global information. If global information is stored using trees, the amount of memory is proportional to the number of leaves of the trees, not to the number of states, facilitating scaling. Reaching a specific region of the state space is difficult when we can only sample the current trajectory since we may not know which actions will get us there. Activities may provide a useful tool to achieve this.

When we learn a DBN model of a factored MDP, what we are ultimately interested in is the ability to learn a behavior that is useful in the task that the factored MDP models. In other words, to evaluate a DBN model of a factored MDP, we should measure the expected future reward of policies learned by algorithms that take advantage of the DBN model. For this purpose, it would be useful to develop a utility measure for DBN models that quickly estimate the expected future reward that resulting policies can achieve. Such a measure could be used to guide action selection in an active learning scheme that attempts to maximize the utility measure of the current DBN model.

6.5 Related work

The Bayesian Information Criterion, or BIC, was first proposed by Schwartz (1978). It is motivated by the posterior probability of a probabilistic model given a set of data instances. The Bayesian Dirichlet metric first appeared in the work of Cooper and Herskovits (1992). Heckerman et al. (1995) modified the Bayesian Dirichlet metric to make it likelihood-equivalent, resulting in the BDe score that we used in this chapter.

Several researchers have studied the problem of learning the structure of Bayesian networks. Buntine (1991) proposed several algorithms that refine the structure of an initial Bayesian network to improve its ability to represent the data. Heckerman et al.

(1995) used their BDe metric to evaluate Bayesian networks and developed a greedy algorithm that improves the metric by repeatedly adding and removing edges in the directed acyclic graph of the network. Friedman et al. (1998) extended structure learning to DBNs, and also considers the case for which the data contains missing values.

Active learning of Bayesian networks was suggested by several researchers around the same time (Murphy, 2001; Tong and Koller, 2001; Steck and Jaakkola, 2002). These techniques perform experiments by clamping a subset of the variables to fixed values and sample over the remaining variables. By varying the fixed values, it is possible to determine whether two or more variables are correlated, thereby deciding which edges to include in the directed acyclic graph of the network. Empirical results indicate that active learning can significantly reduce the time it takes to learn the network structure.

Regularization was first suggested by Poggio and Girosi (1990) in the context of artificial neural networks as a way to enhance the ability to generalize across a set of data instances. Researchers have developed methods that automatically determine optimal regularization parameters (MacKay, 1992). Regularization has also been applied to other areas of machine learning, such as support vector machines (Evgeniou et al., 2000). Our application of regularization to the BIC score is similar to the BIC-δ criterion proposed by Broman (1997).

Recall that the DBN model represents the transition probabilities and expected reward, i.e., the dynamics, of a factored MDP. Several researchers have developed exploration techniques for reinforcement learning that also execute actions with the goal of estimating the dynamics of an MDP as quickly as possible. Dearden et al. (1999) take a Bayesian approach to estimating the dynamics of an MDP, and use a gain metric to select actions that improve the estimate. Model-based interval estimation (Wiering, 1999) uses confidence intervals on the estimated probability distributions

of the MDP to select actions. Interval estimation was first generalized to MDPs by Kaelbling (1993). Strehl and Littman (2004) empirically compared three exploration techniques in reinforcement learning and concluded that model-based interval estimation can dramatically increase the observed learning rate.

CHAPTER 7
CONCLUSION

We have presented a series of novel algorithms that aim at facilitating scaling of reinforcement learning to increasingly large tasks. Our approach to scaling is to simplify a task as much as possible prior to applying reinforcement learning. Since reinforcement learning algorithms have been empirically shown to work better in small tasks, simplifying a task gives reinforcement learning a better chance of successfully approximating a solution. A common theme of our algorithms is the use of hierarchical decomposition and state abstraction to reduce the complexity of learning. Hierarchical decomposition and state abstraction both exploit the structure present in a task to reduce the size of the task prior to learning a policy.

An option is one formalization of the idea of an activity that can take more than a single time-step to complete. Our first algorithm, the H-Tree algorithm, performs state abstraction separately for each option in an existing hierarchy of options. The algorithm uses the U-Tree algorithm (McCallum, 1995) to perform state abstraction by recording transition instances, each of which maps to a unique leaf of a tree. The H-Tree algorithm takes advantage of intra-option state abstraction by using experience recorded during execution of one option to perform state abstraction for other options. The transition instances of the H-Tree algorithm also make it possible to organize memory in a hierarchical way, which reduces the amount of memory necessary to remember key events. Experimental results illustrate the benefits of option-specific state abstraction.

Our next algorithm, VISA, dynamically decomposes factored MDPs into hierarchies of options. The VISA algorithm assumes that the DBN model of a factored MDP is given prior to learning, and uses the DBN model to construct a causal graph describing how state variables are related. The algorithm then searches in the conditional probability trees of the DBN model for exits, i.e., combinations of state variable values and actions that cause the values of other state variables to change. The VISA algorithm introduces an option for each exit and uses sophisticated techniques to construct the components of each option. The result is a hierarchy of options in which the policy of an option selects between options at a lower level in the hierarchy. Experimental results in a series of tasks show that the VISA algorithm significantly outperforms other algorithms based on the DBN model of a task.

Our third algorithm is a method for computing compact models of the options discovered by the VISA algorithm. Existing methods for computing compact option models do not scale well to large tasks. Reinforcement learning algorithms can approximate optimal policies even when the transition probabilities of options are unknown. For this reason, VISA uses reinforcement learning to approximate an optimal policy of each option. If the VISA algorithm had access to compact option models, it could use dynamic programming techniques to compute the option policies without interacting with the environment. Our algorithm constructs partitions with certain properties to reduce the complexity of computing compact option models. The algorithm computes a DBN model for each option identical to the DBN model for primitive actions. This makes it possible to apply existing algorithms that use the DBN model to efficiently approximate option policies.

Finally, we developed an algorithm for active learning of the DBN model in case it is not given prior to learning. Our algorithm constructs the DBN model by growing trees representing the conditional probabilities of the model. The algorithm executes actions according to an active learning scheme that attempts to maximize the total

entropy of distributions at the leaves of the trees. Following the execution of an action, the algorithm records a data instance. When enough data instances map to a leaf, the algorithm evaluates possible refinements at the leaf. The algorithm uses principled measures of the posterior probability of a network to evaluate how the posterior probability changes as a result of a refinement. If at least one refinement increases the posterior probability, the algorithm retains the refinement that results in the largest increase.

We did not perform sensitivity analysis to see how robust our algorithms are to noise. Any time the resulting value of a state variable is uncertain, a DBN learning algorithm could mistakenly interpret the uncertainty as evidence of conditional dependence. As a result, the DBN model would incorrectly contain additional edges. We tested our algorithm in several tasks in which the outcome of actions is stochastic, i.e., the resulting values of state variables are uncertain. In these tasks, the algorithm did not introduce additional edges in the DBN model. However, we did not perform experiments in which we varied the probabilities of resulting state variable values.

It would also be interesting to know how sensitive the VISA algorithm is to errors in the DBN model. What if there are too many or too few edges in the DBNs? We have reason to suspect that the VISA algorithm is quite sensitive to model error. If there are too few edges, the VISA algorithm could fail to detect several subtasks, which would be detrimental to learning the policies of higher-level options. On the other hand, if there are too many edges, the VISA algorithm would likely identify all subtasks. However, additional edges could make the causal graph contain a smaller number of strongly connected components, each of a larger size. Since the complexity of solving a subtask depends on the size of the strongly connected components, additional edges could make the algorithm less efficient at solving the subtasks.

7.1 Future work

Needless to say, there remains a lot of work to be done for reinforcement learning to scale to a wide range of realistic tasks, and this dissertation only represents a step in that direction. It is our firm belief that reinforcement learning remains a viable approach for approximating solutions to sequential decision problems, if supported by the right machinery. Function approximation and algorithms that exploit structure have the potential to provide this machinery as they continue to develop. A fundamental part of exploiting structure is determining the right representation of a task, so developing novel representations is another key aspect of scaling reinforcement learning.

Ravindran (2004) developed several representations that take advantage of symmetry in a task. Executing an action in one part of the state space may have the same symmetrical effect as executing another action in a different part of the state space. In addition, a region of the state space may repeat itself, offering an opportunity to reuse partial policies for acting in that region. Relativized options (Ravindran and Barto, 2003) are defined without an absolute frame of reference and can be used to represent partial policies that are useful in several regions of the state space. Although Ravindran provides a framework for exploiting symmetry, there exist few algorithms that efficiently detect symmetry in a task from experience.

Relational reinforcement learning (Džeroski et al., 1998) combines reinforcement learning with relational representations that enable the use of objects and relations between objects. Relational reinforcement learning opens up a new range of possibilities for hierarchical decomposition and state abstraction. For example, relational representations enable different types of hierarchies, such as classes and subclasses of objects. How to automate hierarchical decomposition and state abstraction in relational reinforcement learning is still an open question.

Mahadevan (2005) recently developed a framework for manifold learning in the context of reinforcement learning. Manifolds are low-dimensional representations of objects in high-dimensional spaces, and can be used to compactly represent the state space of a task. Representation Policy Iteration (Mahadevan, 2005) 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. An interesting research question is how to combine manifold learning with hierarchical decomposition and other ways to exploit structure.

It is unlikely that it is possible to develop a general-purpose algorithm that can approximate solutions to arbitrary sequential decision problems. There will more likely be a range of algorithms that exploit different types of structure to approximate solutions. Which algorithm works best in a specific task depends on the types of structure that are present in the task. Under this scenario, it becomes important to develop algorithms that quickly determine whether or not a task displays a specific type of structure. The causal graph that is part of the VISA algorithm is one such quick way to determine whether a task can easily be decomposed into a hierarchy of activities. If the causal graph of a task does not contain more than one or two strongly connected components, the VISA algorithm will not be able to efficiently decompose the task. Testing a task for structure prior to learning makes is possible to more carefully select an algorithm that is suitable for approximating a solution to the task.

APPENDIX

PROOF OF THEOREM 5.1.2

Equations (5.7) and (5.8) are consistent and have unique solutions if and only if the matrix $M = E - \gamma \sum_{a \in A} \Pi^a P^a (E - B)$ is invertible, i.e., if $\det(M) \neq 0$. $(E - B)$ is a diagonal matrix whose elements are in the range $[0, 1]$. Each element of P^a is in the range $[0, 1]$, and each row of P^a sums to 1. Because of the properties of π, it follows that $\sum_{a \in A} \Pi^a = E$ and that $\sum_{a \in A} \Pi^a P^a$ has the same properties as P^a. Then $\gamma \sum_{a \in A} \Pi^a P^a (E - B)$ is a matrix such that each element is in the range $[0, 1]$ and such that the sum of each row is in the range $[0, 1]$. In other words, M has the following properties, where $n = |S|$:

1. for each $i = 1, \ldots, n$: $\quad 0 \leq m_{ii} \leq 1$,
2. for each $i = 1, \ldots, n$, $j \neq i$: $\quad -m_{ii} \leq m_{ij} \leq 0$,
3. for each $i = 1, \ldots, n$: $\quad 0 \leq \sum_{j=1}^{n} m_{ij} \leq m_{ii}$.

Lemma A.0.1 *An element m_{ii} on the diagonal of M equals 0 if and only if*

1. $\gamma = 1$,
2. $\beta(s_i) = 0$,
3. *for each action $a \in A$ such that $\pi(s_i, a) > 0$, $P(s_i \mid s_i, a) = 1$.*

Proof $m_{ii} = 1 - \gamma \sum_{a \in A} \pi(s_i, a) P(s_i \mid s_i, a)(1 - \beta(s_i))$. The only solution to $m_{ii} = 0$ is $\gamma = 1$, $\beta(s_i) = 0$, and $P(s_i \mid s_i, a) = 1$ for each action $a \in A$ such that $\pi(s_i, a) > 0$.

An option is proper if and only if there is no set of absorbing states S' such that $\beta(s) = 0$ for each state $s \in S'$. A set of states S' is absorbing if and only if the probability of transitioning from any state in S' to any state outside S' is 0. A special case occurs when S' contains a single state s_i such that $\beta(s_i) = 0$ and such that $P(s_i \mid s_i, a)$ for each action $a \in A$ such that $\pi(s_i, a) > 0$. From Lemma A.0.1 it follows that an element m_{ii} on the diagonal of M equals 0 if and only if s_i is an absorbing state such that $\beta(s_i) = 0$. Since no such state exists for a proper option o, we conclude that all elements on the diagonal of M are larger than 0 for a proper option o. Then it is possible to multiply each row of M by $1/m_{ii}$ to obtain a matrix A with the following properties:

1. for each $i = 1, \ldots, n$: $\quad a_{ii} = 1,$
2. for each $i = 1, \ldots, n$, $j \neq i$: $\quad -1 \leq a_{ij} \leq 0,$
3. for each $i = 1, \ldots, n$: $\quad 0 \leq \sum_{j=1}^{n} a_{ij} \leq 1.$

Since matrix A is obtained by multiplying each row of M by a scalar, the determinant of M equals 0 if and only if the determinant of A equals 0. We can write A as

$$A = \begin{pmatrix} 1 & a_{12} & \cdots & a_{1n} \\ a_{21} & 1 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 1 \end{pmatrix} = \begin{pmatrix} - & \mathbf{r}_1 & - \\ - & \mathbf{r}_2 & - \\ & \vdots & \\ - & \mathbf{r}_n & - \end{pmatrix},$$

where $\mathbf{r}_i$ is the ith row of A. It is possible to eliminate an element a_{ij}, $j < i$, by subtracting $a_{ij}\mathbf{r}_j$ from row $\mathbf{r}_i$:

$$\mathbf{r}_i - a_{ij}\mathbf{r}_j = \begin{pmatrix} a_{i1} - a_{ij}a_{j1} & \cdots & a_{ij} - a_{ij} \cdot 1 & \cdots & 1 - a_{ij}a_{ji} & \cdots & a_{in} - a_{ij}a_{jn} \end{pmatrix}.$$

Lemma A.0.2 *$0 \leq 1 - a_{ij}a_{ji} \leq 1$, and $1 - a_{ij}a_{ji} = 0$ if and only if $a_{ij} = a_{ji} = -1$.*

Proof Follows immediately from the properties of A.

Lemma A.0.3 *If $1 - a_{ij}a_{ji} > 0$, elimination of a_{ij} preserves the properties of A.*

Proof Since $1 - a_{ij}a_{ji} > 0$, we can multiply $\mathbf{r}_i - a_{ij}\mathbf{r}_j$ by $1/(1 - a_{ij}a_{ji})$:

$$\bar{\mathbf{r}}_i = \frac{1}{1 - a_{ij}a_{ji}}[\mathbf{r}_i - a_{ij}\mathbf{r}_j] = \begin{pmatrix} \frac{a_{i1} - a_{ij}a_{j1}}{1 - a_{ij}a_{ji}} & \cdots & 0 & \cdots & 1 & \cdots & \frac{a_{in} - a_{ij}a_{jn}}{1 - a_{ij}a_{ji}} \end{pmatrix}.$$

It follows immediately that element i of row $\bar{\mathbf{r}}_i$ equals $(1 - a_{ij}a_{ji})/(1 - a_{ij}a_{ji}) = 1$ and that element j equals $(a_{ij} - a_{ij} \cdot 1)/(1 - a_{ij}a_{ji}) = 0$. For each $k = 1, \ldots, n$, $k \neq i, j$, compute bounds on element k of $\bar{\mathbf{r}}_i$:

$$\begin{aligned}
\frac{a_{ik} - a_{ij}a_{jk}}{1 - a_{ij}a_{ji}} &\leq \frac{a_{ik} - 0}{1 - a_{ij}a_{ji}} \leq \frac{0 - 0}{1 - a_{ij}a_{ji}} = 0, \\
\frac{a_{ik} - a_{ij}a_{jk}}{1 - a_{ij}a_{ji}} &= \frac{1 - a_{ij}a_{ji} + a_{ik} - a_{ij}a_{jk} - (1 - a_{ij}a_{ji})}{1 - a_{ij}a_{ji}} = \\
&= \frac{1 + a_{ik} - a_{ij}(a_{ji} + a_{jk})}{1 - a_{ij}a_{ji}} - 1 \geq \\
&\geq \frac{1 + a_{ik} + a_{ij}}{1 - a_{ij}a_{ji}} - 1 \geq \frac{1 - 1}{1 - a_{ij}a_{ji}} - 1 = -1.
\end{aligned}$$

Also compute bounds on the sum of the elements of $\bar{\mathbf{r}}_i$:

$$\begin{aligned}
\sum_{k=1}^{n} \frac{a_{ik} - a_{ij}a_{jk}}{1 - a_{ij}a_{ji}} &= \sum_{k=1}^{j-1} \frac{a_{ik} - a_{ij}a_{jk}}{1 - a_{ij}a_{ji}} + \frac{a_{ij} - a_{ij} \cdot 1}{1 - a_{ij}a_{ji}} + \sum_{k=j+1}^{i-1} \frac{a_{ik} - a_{ij}a_{jk}}{1 - a_{ij}a_{ji}} + \\
&+ \frac{1 - a_{ij}a_{ji}}{1 - a_{ij}a_{ji}} + \sum_{k=i+1}^{n} \frac{a_{ik} - a_{ij}a_{jk}}{1 - a_{ij}a_{ji}} \leq \\
&\leq 0 + 0 + 0 + 1 + 0 = 1, \\
\sum_{k=1}^{n} \frac{a_{ik} - a_{ij}a_{jk}}{1 - a_{ij}a_{ji}} &= \frac{1}{1 - a_{ij}a_{ji}} \left[\sum_{k=1}^{n} a_{ik} - a_{ij} \sum_{k=1}^{n} a_{jk} \right] \geq \frac{0 + 0}{1 - a_{ij}a_{ji}} = 0.
\end{aligned}$$

It follows that row $\bar{\mathbf{r}}_i$ satisfies the properties of A.

From Lemma A.0.2 and Lemma A.0.3 it follows that the properties of A are preserved under elimination unless the element on the diagonal equals 0. We can compute the determinant of A by repeatedly performing elimination until A is an upper triangular matrix. If any element on the diagonal becomes 0 during elimination, $\det(A) = 0$. Otherwise, the determinant of A equals the inverse of the product of the coefficients by which we multiplied rows during elimination. Since each coefficient is larger than 0, it follows that $\det(A) > 0$.

Lemma A.0.4 *Let $C = \{c_1, \ldots, c_m\}$ be a set of m indices, and let $S(C, \mathbf{r}_i) = \sum_{k=1}^{m} a_{ic_k}$ be the sum of elements of row $\mathbf{r}_i$ whose column indices are elements of C. Assume that $i \in C$ and that $S(C, \bar{\mathbf{r}}_i) = 0$ after elimination of an element a_{ij}, $j < i$. Then $S(C \cup \{j\}, \mathbf{r}_i) = 0$ and $S(C \cup \{j\}, \mathbf{r}_j) = 0$ prior to elimination of a_{ij}.*

Proof When we eliminate an element a_{ij}, $j < i$, the sum of elements of row $\bar{\mathbf{r}}_i$ whose column indices are elements of C is

$$\begin{aligned} S(C, \bar{\mathbf{r}}_i) &= S(C, \bar{\mathbf{r}}_i) + 0 = \\ &= \sum_{k=1}^{m} \frac{a_{ic_k} - a_{ij}a_{jc_k}}{1 - a_{ij}a_{ji}} + \frac{a_{ij} - a_{ij} \cdot 1}{1 - a_{ij}a_{ji}} = \\ &= \frac{1}{1 - a_{ij}a_{ji}} \left[\left(\sum_{k=1}^{m} a_{ic_k} + a_{ij} \right) - a_{ij} \left(\sum_{k=1}^{m} a_{jc_k} + 1 \right) \right]. \end{aligned}$$

Since i is one of the indices in C, it follows from the properties of A that $S(C, \bar{\mathbf{r}}_i) = 0$ if and only if $\sum_{k=1}^{m} a_{ic_k} + a_{ij} = 0$ and either $a_{ij} = 0$ or $\sum_{k=1}^{m} a_{jc_k} + 1 = 0$. If $a_{ij} = 0$, there was no reason to perform elimination, so it follows that $S(C \cup \{j\}, \mathbf{r}_i) = \sum_{k=1}^{m} a_{ic_k} + a_{ij} = 0$ and that $S(C \cup \{j\}, \mathbf{r}_j) = \sum_{k=1}^{m} a_{jc_k} + 1 = 0$.

Lemma A.0.5 *If $S(C, \mathbf{r}_k) = 0$ for each row $\mathbf{r}_k$, $k \in C$, after elimination of an element a_{ij}, $j \notin C$, in row $\mathbf{r}_i$, $i \in C$, it follows that $S(C \cup \{j\}, \mathbf{r}_k) = 0$ for each row $\mathbf{r}_k$, $k \in C \cup \{j\}$ prior to elimination of a_{ij}.*

Proof If $S(C, \mathbf{r}_i) = 0$ following elimination of a_{ij}, it follows from Lemma A.0.4 that $S(C \cup \{j\}, \mathbf{r}_i) = 0$ and that $S(C \cup \{j\}, \mathbf{r}_j) = 0$ prior to elimination of a_{ij}. For $k \in C - \{i\}$, $S(C \cup \{j\}, \mathbf{r}_k) = S(C, \mathbf{r}_k) + a_{kj} = a_{kj}$. Since $a_{kj} \leq 0$ and $S(C \cup \{j\}, \mathbf{r}_k) \geq 0$, it follows that $S(C \cup \{j\}, \mathbf{r}_k) = a_{kj} = 0$.

Lemma A.0.6 *If* $\det(A) = 0$*, it is possible to rearrange the rows and columns of* A *to obtain*

$$\begin{pmatrix} X & 0 \\ Y & Z \end{pmatrix},$$

where X *is a* $k \times k$ *matrix such that for each* $i = 1, \ldots, k$, $\sum_{j=1}^{k} x_{ij} = 0$.

Proof If $\det(A) = 0$, there exists $i, j < i$ such that a_{ii} becomes 0 during elimination of a_{ij}. From Lemma A.0.2 it follows that $a_{ij} = a_{ji} = -1$ prior to elimination of a_{ij}, so $S(\{i, j\}, \mathbf{r}_i) = a_{ij} + a_{ii} = -1 + 1 = 0$ and $S(\{i, j\}, \mathbf{r}_j) = a_{jj} + a_{ji} = 1 - 1 = 0$. Let $C = \{i, j\}$. Recursively find each index l such that elimination of element a_{kl} occured prior to this round in row $\mathbf{r}_k$, $k \in C$. Then it follows from Lemma A.0.5 that $S(C \cup \{l\}, \mathbf{r}_k) = 0$ for each $k \in C \cup \{l\}$ prior to elimination of a_{kl}. Add each such index l to C. Prior to elimination of any element, it is possible to rearrange the rows and columns of A to obtain

$$\begin{pmatrix} a'_{11} & \cdots & a'_{1m} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a'_{m1} & \cdots & a'_{mm} & 0 & \cdots & 0 \\ a'_{(m+1)1} & \cdots & a'_{(m+1)m} & a'_{(m+1)(m+1)} & \cdots & a'_{(m+1)n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a'_{n1} & \cdots & a'_{nm} & a'_{n(m+1)} & \cdots & a'_{nn} \end{pmatrix},$$

where the first m rows and columns are those whose indices are elements of C. Since $S(C, \mathbf{r}_k) = 0$ for each row $\mathbf{r}_k$, $k \in C$, it follows that the sum of row $\mathbf{r}_k$ equals 0 and that for each $l \notin C$, element a_{kl} equals 0.

From the definition of M it follows that it is only possible to rearrange the rows and columns to obtain

$$\begin{pmatrix} X & 0 \\ Y & Z \end{pmatrix},$$

such that the sum of each row of X equals 0, if there is an absorbing set of states S' such that $\beta(s) = 0$ for each state $s \in S'$ and if $\gamma = 1$. For a proper option o, it is not possible to rearrange M that way. Since the sum of one row of M equals 0 if and only if the sum of the same row of A equals 0, it is not possible to rearrange A that way either. It follows from the contrapositive of Lemma A.0.6 that $\det(A) \neq 0$, which also means that $\det(M) \neq 0$. This concludes the proof of Theorem 5.1.2.

BIBLIOGRAPHY

Andre, D. and Russell, S. (2002). State Abstraction for Programmable Reinforcement Learning Agents. *Proceedings of the National Conference on Artificial Intelligence*, 18:119–125.

Åström, K. (1965). Optimal control of Markov decision processes with incomplete state estimation. *Journal of Mathematical Analysis and Applications*, 10:174–205.

Barto, A., Singh, S., and Chentanez, N. (2004). Intrinsically Motivated Learning of Hierarchical Collections of Skills. *Proceedings of the International Conference on Development and Learning*, 3:112–119.

Bellman, R. (1956). A problem in the sequential design of experiments. *Sankhya*, 16:221–229.

Bellman, R. (1957). A Markov decision process. *Journal of Mathematical Mechanics*, 6:679–684.

Boutilier, C. and Dearden, R. (1994). Using Abstractions for Decision-Theoretic Planning with Time Constraints. *Proceedings of the National Conference on Artificial Intelligence*, 12:1016–1022.

Boutilier, C., Dearden, R., and Goldszmidt, M. (1995). Exploiting structure in policy construction. *Proceedings of the International Joint Conference on Artificial Intelligence*, 14:1104–1113.

Boutilier, C., Dearden, R., and Goldszmidt, M. (2000). Stochastic Dynamic Programming with Factored Representations. *Artificial Intelligence*, 121(1):49–107.

Bradtke, S. and Duff, M. (1995). Reinforcement learning methods for continuous-time Markov decision problems. *Advances in Neural Information Processing Systems*, 7:393–400.

Broman, K. (1997). *Identifying quantitative trait loci in experimental crosses*. Ph.D Thesis, University of California, Berkeley, USA.

Bulitko, V., Sturtevant, N., and Kazakevich, M. (2005). Speeding Up Learning in Real-time Search via Automatic State Abstraction. *Proceedings of the National Conference on Artificial Intelligence*, 20.

Buntine, W. (1991). Theory refinement on Bayesian networks. *Proceedings of Uncertainty in Artificial Intelligence*, 7:52–60.

Chapman, D. and Kaelbling, L. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. *Proceedings of the International Joint Conference on Artificial Intelligence*, 12:726–731.

Chickering, D., Geiger, D., and Heckerman, D. (1995). Learning Bayesian networks: search methods and experimental results. *Proceedings of Artificial Intelligence and Statistics*, 5:112–128.

Cockett, J. (1985). *Decision expression optimization*. Technical report, University of Tennessee, Knoxville, USA.

Cooper, G. and Herskovits, E. (1992). A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9:309–347.

Crites, R. and Barto, A. (1996). Improving elevator performance using reinforcement learning. *Advances in Neural Information Processing Systems*, 8:1017–1023.

Dean, T. and Givan, R. (1997). Model minimization in Markov decision processes. *Proceedings of the National Conference on Artificial Intelligence*, 14:106–111.

Dean, T. and Kanazawa, K. (1989). A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150.

Dearden, R., Friedman, N., and Andre, D. (1999). Model based Bayesian Exploration. *Proceedings of Uncertainty in Artificial Intelligence*, 15:150–159.

Dietterich, T. (2000a). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303.

Dietterich, T. (2000b). State Abstraction in MAXQ Hierarchical Reinforcement Learning. *Advances in Neural Information Processing Systems*, 12:994–1000.

Digney, B. (1996). Emergent hierarchical control structures: Learning reactive/hierarchical relationships in reinforcement environments. *From animals to animats*, 4:363–372.

Džeroski, S., de Raedt, L., and Blockeel, H. (1998). Relational reinforcement learning. *Proceedings of the International Conference on Machine Learning*, 15:136–143.

Evgeniou, T., Pontil, M., and Poggio, T. (2000). Regularization Networks and Support Vector Machines. *Advances in Computational Mathematics*, 13(1):1–50.

Feng, Z., Hansen, E., and Zilberstein, S. (2003). Symbolic generalization for on-line planning. *Proceedings of Uncertainty in Artificial Intelligence*, 19:209–216.

Fikes, R. and Nilsson, N. (1971). Strips: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189–208.

Friedman, N., Murphy, K., and Russell, S. (1998). Learning the structure of dynamic probabilistic networks. *Proceedings of Uncertainty in Artificial Intelligence*, 14:139–147.

Ghavamzadeh, M. and Mahadevan, S. (2001). Continuous-Time Hierarchical Reinforcement Learning. *Proceedings of the International Conference on Machine Learning*, 18:186–193.

Guestrin, C., Koller, D., and Parr, R. (2001). Max-norm Projections for Factored MDPs. *International Joint Conference on Artificial Intelligence*, 17:673–680.

Heckerman, D., Geiger, D., and Chickering, D. (1995). Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20:197–243.

Helmert, M. (2004). A planning heuristic based on causal graph analysis. *Proceedings of the International Conference on Automated Planning and Scheduling*, 14:161–170.

Hengst, B. (2002). Discovering Hierarchy in Reinforcement Learning with HEXQ. *Proceedings of the International Conference on Machine Learning*, 19:243–250.

Hernandez-Gardiol, N. and Mahadevan, S. (2001). Hierarchical Memory-Based Reinforcement Learning. *Advances in Neural Information Processing Systems*, 13:1047–1053.

Hoey, J., St-Aubin, R., Hu, A., and Boutilier, C. (1999). Spudd: Stochastic Planning using Decision Diagrams. *Proceedings of Uncertainty in Artificial Intelligence*, 15:279–288.

Howard, R. (1960). *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, USA.

Jonsson, A. and Barto, A. (2001). Automated State Abstractions for Options Using the U-Tree Algorithm. *Advances in Neural Information Processing Systems*, 13:1054–1060.

Jonsson, A. and Barto, A. (2005). A Causal Approach to Hierarchical Decomposition of Factored MDPs. *Proceedings of the International Conference on Machine Learning*, 22.

Kaelbling, L. (1993). *Learning in embedded systems.* MIT Press, Cambridge, USA.

Kearns, M. and Koller, D. (1999). Efficient Reinforcement Learning in Factored MDPs. *Proceedings of the International Joint Conference on Artificial Intelligence*, 16:740–747.

MacKay, D. (1992). Bayesian Interpolation. *Neural Computation*, 4(3):415–447.

Mahadevan, S. (2005). Representation Policy Iteration. *Proceedings of Uncertainty in Artificial Intelligence*, 21.

Mannor, S., Menache, I., Hoze, A., and Klein, U. (2004). Dynamic abstraction in reinforcement learning via clustering. *Proceedings of the International Conference on Machine Learning*, 21:560–567.

McCallum, A. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. Ph.D. Thesis, Computer Science Department, University of Rochester, USA.

McGovern, A. and Barto, A. (2001). Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density. *Proceedings of the International Conference on Machine Learning*, 18:361–368.

Menache, I., Mannor, S., and Shimkin, N. (2002). Q-Cut – Dynamic Discovery of Sub-Goals in Reinforcement Learning. *Proceedings of the European Conference on Machine Learning*, 13:295–306.

Moore, A. and Atkeson, C. (1995). The Parti-game Algorithm for Variable Resolution Reinforcement Learning in Multidimensional State-spaces. *Machine Learning*, 21:199–233.

Munos, R. and Moore, A. (1999). Variable resolution discretization for high-accuracy solutions of optimal control problems. *Proceedings of the International Joint Conference on Artificial Intelligence*, 16:1348–1355.

Murphy, K. (2001). *Active Learning of Causal Bayes Net Structure*. Technical Report, University of California, Berkeley, USA.

Parr, R. and Russell, S. (1998). Reinforcement Learning with Hierarchies of Machines. *Advances in Neural Information Processing Systems*, 10:1043–1049.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, USA.

Pickett, M. and Barto, A. (2002). Policyblocks: An Algorithm for Creating Useful Macro-Actions in Reinforcement Learning. *Proceedings of the International Conference on Machine Learning*, 19:506–513.

Poggio, T. and Girosi, F. (1990). Regularization Algorithms for Learning that are Equivalent to Multilayer Networks. *Science*, 247:978–982.

Puterman, M. (1990). Markov decision processes. *Handbooks in Operations Research and Management Science*, 2:331–434.

Puterman, M. (1994). *Markov Decision Processes*. Wiley Interscience, New York, USA.

Puterman, M. and Brumelle, S. (1979). On the convergence of policy iteration in stationary dynamic programming. *Mathematics of Operations Research*, 4:60–69.

Ravindran, B. (2004). *An Algebraic Approach to Abstraction in Reinforcement Learning*. Ph.D. Thesis, Department of Computer Science, University of Massachusetts, Amherst, USA.

Ravindran, B. and Barto, A. (2003). Relativized Options: Choosing the Right Transformation. *Proceedings of the International Conference on Machine Learning*, 18:1011–1016.

Schwartz, G. (1978). Estimating the dimension of a model. *Annals of Statistics*, 6:461–464.

Şimşek, Ö. and Barto, A. (2004). Using relative novelty to identify useful temporal abstractions in reinforcement learning. *Proceedings of the International Conference on Machine Learning*, 21:751–758.

Şimşek, Ö., Wolfe, A., and Barto, A. (2005). Identifying useful subgoals in reinforcement learning by local graph partitioning. *Proceedings of the International Conference on Machine Learning*, 22.

Smith, J. (1971). Markov Decisions on a Partitioned State Space. *IEEE Transactions on Systems, Man, and Cybernetics*, 1:55–60.

Steck, H. and Jaakkola, T. (2002). Unsupervised Active Learning in Large Domains. *Proceedings of Uncertainty in Artificial Intelligence*, 18:469–476.

Strehl, A. and Littman, M. (2004). An Empirical Evaluation of Interval Estimation for Markov Decision Processes. *Proceedings of the International Conference on Tools with Artificial Intelligence*, 16:128–135.

Sutton, R. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44.

Sutton, R. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems*, 8:1038–1044.

Sutton, R. and Barto, A. (1998). *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, USA.

Sutton, R., Precup, D., and Singh, S. (1998). Intra-Option Learning about Temporally Abstract Actions. *Proceedings of the International Conference on Machine Learning*, 15:556–564.

Sutton, R., Precup, D., and Singh, S. (1999). Between MDPs and Semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211.

Tesauro, G. (1994). Td-Gammon, a self-teaching backgammon program achieves master-level play. *Neural Computation*, 6:215–219.

Thrun, S. and Schwartz, A. (1996). Finding structure in reinforcement learning. *Advances in Neural Information Processing Systems*, 8:385–392.

Tong, S. and Koller, D. (2001). Active learning for parameter estimation in Bayesian networks. *Advances in Neural Information Processing Systems*, 13:647–653.

Watkins, C. (1989). *Learning from delayed rewards.* Ph.D. Thesis, Psychology Department, University of Cambridge, UK.

Watkins, C. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8:279–292.

Wiering, M. (1999). *Explorations in efficient reinforcement learning.* Ph.D Thesis, University of Amsterdam, Netherlands.

Zhang, W. and Dietterich, T. (1995). A Reinforcement Learning Approach to Job-shop Scheduling. *Proceedings of the International Joint Conference on Artificial Intelligence*, 14:1114–1120.