Adaptive Modelling and Planning for Learning Intelligent Behaviour

Mykel J. Kochenderfer

Doctor of Philosophy

Institute of Perception, Action and Behaviour
School of Informatics
University of Edinburgh

2006

Abstract

An intelligent agent must be capable of using its past experience to develop an understanding of how its actions affect the world in which it is situated. Given some objective, the agent must be able to effectively use its understanding of the world to produce a plan that is robust to the uncertainty present in the world. This thesis presents a novel computational framework called the Adaptive Modelling and Planning System (AMPS) that aims to meet these requirements for intelligence.

The challenge of the agent is to use its experience in the world to generate a model. In problems with large state and action spaces, the agent can generalise from limited experience by grouping together similar states and actions, effectively partitioning the state and action spaces into finite sets of regions. This process is called abstraction. Several different abstraction approaches have been proposed in the literature, but the existing algorithms have many limitations. They generally only increase resolution, require a large amount of data before changing the abstraction, do not generalise over actions, and are computationally expensive. AMPS aims to solve these problems using a new kind of approach.

AMPS splits and merges existing regions in its abstraction according to a set of heuristics. The system introduces splits using a mechanism related to supervised learning and is defined in a general way, allowing AMPS to leverage a wide variety of representations. The system merges existing regions when an analysis of the current plan indicates that doing so could be useful. Because several different regions may require revision at any given time, AMPS prioritises revision to best utilise whatever computational resources are available. Changes in the abstraction lead to changes in the model, requiring changes to the plan. AMPS prioritises the planning process, and when the agent has time, it replans in high-priority regions. This thesis demonstrates the flexibility and strength of this approach in learning intelligent behaviour from limited experience.

Acknowledgements

First of all, I must thank my primary advisor Gillian Hayes for her enthusiasm and encouragement while I undertook this work. I have benefited tremendously from our discussions over these past three years. I would also like to acknowledge the support of my two other advisors, John Levine and Austin Tate. Michael Littman and Sethu Vijayakumar served as my examiners, and their feedback during and after my defense was invaluable.

Special thanks are due to Nils Nilsson, my mentor while I was an undergraduate at Stanford University. He has greatly influenced me as a person and as a researcher, and he continues to serve as my inspiration. Many of the core ideas of this thesis have emerged from early discussions with Nils.

I am grateful for the feedback from other doctoral students and researchers at the AAAI/SIGART Doctoral Consortium in 2005. I am also indebted to the anonymous reviewers of my papers describing the Adaptive Modelling and Planning System (Kochenderfer, 2005; Kochenderfer and Hayes, 2005a,b).

Several members of the Institute of Perception, Action and Behaviour have been tremendously helpful in providing feedback, including Jay Bradley, Paul Crook, Bob Fisher, George Konidaris, Graham McNeill, Hugo Rosano, Matthew Szenher, Tim Taylor, Marc Toussaint, and Jose Vasquez.

Finally, I wish to thank my wife, Mary Anne Kochenderfer. She has helped me proofread numerous drafts of this thesis and helped me stay disciplined. She supported me all along the way, even though she was entrenched with her own thesis work and being a mother to our new baby, Emma.

Funding for this research comes in part from the United Kingdom Overseas Research Award and scholarships from the School of Informatics and the American Friends of the University of Edinburgh.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Mykel J. Kochenderfer
Edinburgh
24 July 2006

To my wife and in memory of my dad.

Table of Contents

Chapter 1

Introduction

One of the most fundamental problems in artificial intelligence is the control of an agent situated in a world. An *agent* is an entity that perceives and acts in an environment in pursuit of some goal. This thesis presents a novel framework, called the *Adaptive Modelling and Planning System* (AMPS), as a way to generate intelligent behaviour in real time through reinforcement learning. This chapter discusses the challenges in building an effective reinforcement learning agent, the approach taken in this thesis, and a sampling of problem domains. The final sections conclude with a summary of contributions and an overview of the thesis.

1.1 Reinforcement Learning

The objective of this thesis is to explore ways of achieving *intelligent* behaviour in artificial agents. According to the *Oxford English Dictionary* (edited by Simpson and Weiner, 1989), for an agent to be intelligent it must be "able to vary its behaviour in response to varying situations and requirements and past experience." This thesis explores a novel approach to achieving these qualities of intelligence from the perspective of reinforcement learning.

In reinforcement learning problems, an agent receives reward while interacting with the world. The objective of the agent is to maximise its expected accumulation of reward, or *return*. It is generally assumed that the state of the world changes in response to the actions taken by the agent according to some model. Typically, the agent does not possess a complete and accurate model of the world, and so it must leverage its experience using a model-based or model-free approach.

Model-based approaches calculate the expected return based on a model of

the system dynamics estimated from experience. *Model-free* approaches, on the other hand, directly estimate return. Model-based approaches generally perform significantly better than model-free approaches, requiring less time and experience in the world. Since it is desirable to construct agents that learn as quickly as possible, this thesis focuses on model-based learning.

Modelling refers to estimating a model of the world from experience. Models are used to predict reward and the dynamics of the world in response to the behaviour of the agent. Depending upon the application, different classes of models might be appropriate. For example, in some finance applications, the dynamics might be represented by stochastic differential equations, whereas some factory optimisation problems can be modelled by semi-Markov decision processes.

If the agent possesses an estimate of the world model, it can use a computational process called *planning* to compute the expected return and decide upon an intelligent course of action. The appropriate planning mechanism for a particular problem depends on the class of models under consideration. Some planning approaches produce *reactive plans*, which are complete mappings from states to actions. Other planning approaches produce sequences of actions to be taken from the current state. This thesis focuses on reactive planning since it is better suited for producing real-time behaviour in dynamic environments.

1.2 Challenges

There are several issues that make integrating modelling and planning for real-time control difficult. This section introduces some of the major challenges associated with generalisation, adaptation, abstraction, efficiency, and representation.

Perhaps the most significant issue in reinforcement learning is *generalisation*. If the state and action spaces are large, which is typically the case in real-world problems, then the agent must generalise its experience about specific states and actions and reason about states and actions with which it has no experience. Since generalisation is one of the key issues in applying reinforcement learning methods to real-world problems, much recent work focuses on this issue. Chapter 4 surveys a wide variety of previous approaches to generalisation, identifying some of their key limitations that this thesis aims to address.

Abstraction is one way to achieve generalisation. Instead of treating states and actions individually, abstraction involves grouping together states and ac-

tions and treating them collectively. One abstract state might consist of many different states in the state space, and one abstract action might consist of many different actions in the action space. It is still an open question how to go about partitioning the state and action spaces effectively. It is also not clear how to revise these partitions efficiently as the agent acquires experience.

Generalisation is intimately connected to representation. Any given problem may be represented in a variety of ways, some more natural than others. Successful generalisation depends on leveraging whatever structure may be gleaned from the state and action space representation.

Most reinforcement learning algorithms assume a particular representation, such as an attribute-value, inner-product space, or relational representation. However, this dependence upon representation is not desirable since a change in domain can require a complete change in representation. A poor choice of representation can result in poor performance.

1.3 Approach

This thesis presents a novel framework, called the *Adaptive Modelling and Planning System* (AMPS), that integrates modelling and planning for the generation of intelligent reactive behaviour. The system addresses the challenges mentioned in the previous section and is suitable for a broad class of problems. This section presents the approach taken in AMPS.

In order to learn efficiently in a world with little prior knowledge, AMPS constructs a model from experience. AMPS estimates the parameters of a semi-Markov decision process to model the environmental dynamics. A semi-Markov decision process models probabilistic state transitions, durations, and rewards, making it applicable to a wide variety of problems.

When the state and action spaces of a problem are large, generalisation is necessary, and AMPS accomplishes this goal through abstraction. Abstraction involves partitioning the state and action spaces into regions. Each abstract state consists of a set of states, and each abstract action consists of a set of actions. With the state and action spaces partitioned into a finite set of abstract states and actions, AMPS can estimate the parameters of the model from experience and then use this model to generate a plan.

Figure 1.1 summarises some of the major processes in AMPS and serves as a

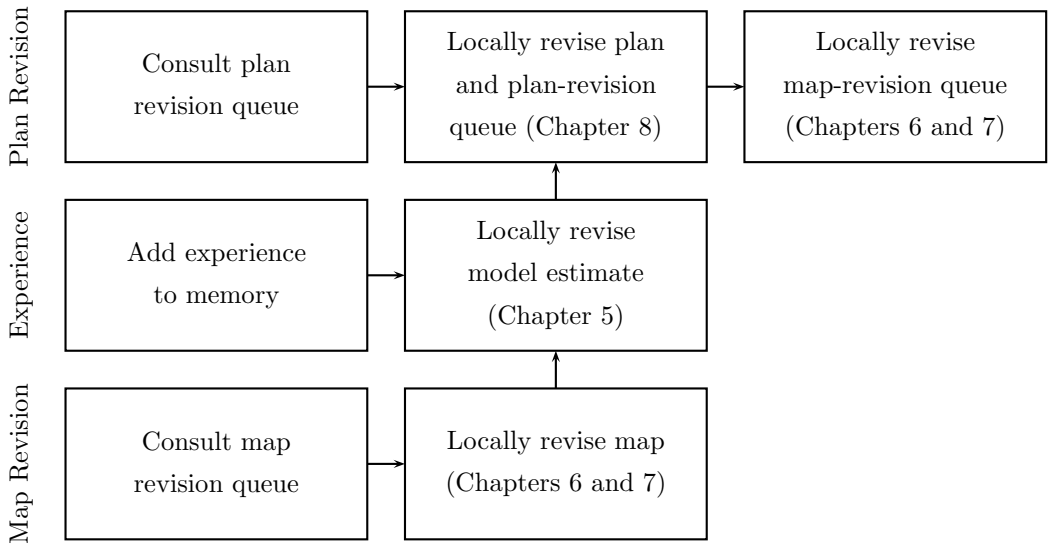


Figure 1.1: The major processes in AMPS.

map of the major components discussed in the chapters of this thesis. Not shown in this figure is the action-selection process, which involves consulting the current plan and applying some exploration policy. The remainder of this section will discuss the components illustrated in Figure 1.1 in greater detail.

Since it is impractical in most complex problems to generate a new plan from scratch whenever the model changes, AMPS maintains a priority queue of the regions most in need of plan revision. Whenever the agent has time to revise its plan, it consults the plan-revision queue and performs local plan revision in the highest priority regions. The procedure for assigning priorities and revising plans is discussed in detail in Chapter 8.

As the agent interacts with the world, its experiences are recorded in memory in order to estimate model parameters, which will be discussed in Chapter 5. As each experience is added to memory, the model is locally revised, leading to local changes in the plan and the plan-revision queue. If the agent is running low on memory, old experience may be removed and the model updated accordingly.

Map revision involves adapting how the state and action spaces are partitioned through either splitting or merging existing regions. Like plan revision, map revision is prioritised. When the agent has time, the map-revision queue is consulted and the highest priority revision is performed. After the revision is performed, the agent updates the model to reflect the changes to the abstraction.

The changes to the model result in changes to the plan as well as the plan-revision and map-revision queues.

Deciding how and when to split and merge regions is discussed in detail in Chapters 6 and 7. Briefly, AMPS splits regions when there are observed differences in the expected return along transitions between state regions. Failure, such as becoming "stuck" in a particular state, also indicates that a region should be split. Splitting, especially when based on limited experience, can lead to overly complex models and representations. Hence, it is important to merge regions when possible. AMPS uses heuristics for deciding when merging is necessary.

The split and merge operations in AMPS are defined abstractly, allowing flexibility in the choice of data structures and algorithms for maintaining the partitions of the state and action spaces. This thesis investigates two approaches based on supervised learning. The first approach uses a decision graph, which partitions the state and action spaces into regions through a series of tests. The second approach uses a distance metric, which partitions the state and action spaces according to proximity to stored instances. Further detail of these two approaches is reserved for Sections 6.4 and 6.5.

1.4 Problem Domains

This section introduces the problem domains used throughout this thesis. These domains were chosen to evaluate how well AMPS can cope with different kinds of problem structures and representations. Considered are problems with both discrete and continuous state and action spaces.

1.4.1 Goal World

The simplest domain, considered primarily for illustrative purposes in this thesis, is Goal World. It has a continuous state space and a discrete action space. In this problem, the agent is permitted to rotate and translate at a fixed velocity with the objective of reaching a goal in a random location. The agent perceives the state of the world as a vector

$$(x, y, \theta, \textit{goal-}x, \textit{goal-}y)\,,$$

where x and y are the coordinates of the centre of the agent, θ is the orientation of the agent, and *goal-*x and *goal-*y are the coordinates of the centre of the goal.

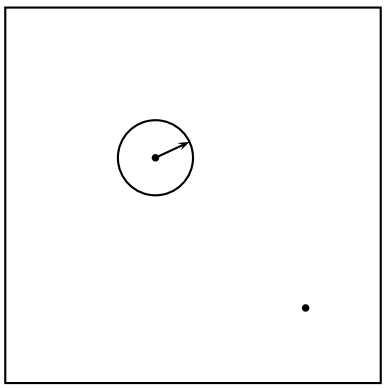


Figure 1.2: The Goal World domain. The arrow indicates the orientation of the agent and the dot in the bottom-right corner is the goal.

The actions available to the agent include Rotate-Left, Rotate-Right, and Move-Forward. The agent may translate only in the direction of its current orientation. When the centre of the agent comes within a small threshold of the goal, the agent receives unit reward. Figure 1.2 illustrates a possible configuration.

1.4.2 Taxi World

In the Taxi World domain, the agent drives a taxi, picking up passengers and delivering them to their destination. The state of the world is represented by

$$(\textit{in-taxi}, \textit{taxi-}x, \textit{taxi-}y, \textit{passenger-}x, \textit{passenger-}y, \textit{destination-}x, \textit{destination-}y)\,,$$

where the first element indicates whether the passenger is in the taxi and the others represent the x and y coordinates of the taxi, passenger, and destination. Figure 1.3 illustrates a possible state of the world. Although the figure shows a 10×10 grid, the experiments use larger grids. The actions available to the agent include Move-North, Move-South, Move-East, Move-West, Pick-Up, and Drop-Off. These actions have their intended effect most of the time, but with some small probability they behave differently. The agent receives unit reward when the passenger reaches its destination.

1.4.3 Corner World

The Corner World domain has a continuous state and action space. The agent starts at a random location on the starting line and maneuvers along an L-shaped

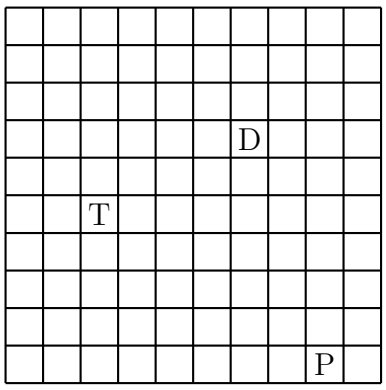


Figure 1.3: The Taxi World domain. In the grid, *T* represents the position of the taxi, *P* represents the position of the passenger, and *D* represents the destination. The experiments use a larger grid than the one shown above.

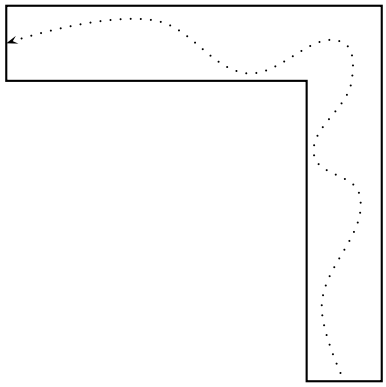


Figure 1.4: The Corner World domain. The dotted line shows a sample trajectory. The experiments use a narrower track than the one shown above.

track to the finish line. Figure 1.4 shows a sample track with a sample trajectory. A state is represented by a tuple (x, y) corresponding to the location of the agent. The agent may translate in any direction at some fixed velocity. The agent samples the environment and makes control decisions at some frequency. As the agent interacts with the world, the position of the agent is perturbed by an amount selected from a normal distribution.

The Goal World, Taxi World, and Corner World domains are presented in this introductory chapter because they are used to illustrate various points throughout this thesis. Chapter 10 uses these domains to analyse the behaviour of AMPS and compare its properties and performance with other approaches.

1.5 Contributions

Contributions are highlighted in the chapters in which they are presented, and Chapter 11 summarises the specific contributions made throughout this thesis. This section provides a brief, high-level overview of the primary contributions.

This thesis presents a general framework for integrating modelling and planning for reinforcement problems. An implementation of this framework demonstrates that it can produce competent behaviour in large stochastic domains with incomplete knowledge of the system dynamics and very little experience.

An important contribution of this framework is the view that both modelling and planning can be performed as interleaved prioritised processes. This contribution is significant because it allows the model and plan to be adapted when the agent has only short, periodic time slices available for computation.

This thesis advances the view that because representation is critical to generalisation, the details of the representation should be abstracted away and the mechanism for interacting with the representation should be self-contained in an independent module. This module can be engineered specifically to the domain, without having to modify the implementation of the general system.

The ideas of how to keep the mechanism that handles the underlying representation self-contained are novel. One of the ideas is based on concepts borrowed from supervised learning, although in the proposed framework the labelled examples are generated by an unsupervised process. Since these mechanisms need to be tailored to the specific representation, this thesis explores ways to automate this process and provides algorithms and tools to quickly build such mechanisms.

The approach suggested in this thesis addresses many of the limitations of existing abstraction algorithms. Most existing algorithms only increase perceptual resolution, but this thesis shows how to both increase and decrease resolution dynamically as the agent accumulates experience. Unlike other abstraction algorithms, AMPS may generalise over both the state and action space.

Some of the contributions in this thesis combine or extend ideas explored by others. Although many of the ideas in this thesis are new, some ideas are borrowed from research done both inside and outside the area of reinforcement learning. The method used for adapting abstractions in AMPS combines and modifies ideas found elsewhere. The planning process in AMPS is an adaptation of other work on prioritised value iteration, extended to continuous-time problems.

1.6 Overview

This thesis presents AMPS as a way to solve complex reinforcement learning problems in real time. This chapter has introduced the class of problems to be solved, discussed the challenges AMPS aims to overcome, explained the system architecture and problems to be solved, and outlined the contributions. The remainder of this thesis proceeds as follows.

Chapter 2 provides an overview of the various approaches taken in the field of artificial intelligence for building autonomous agents. The purpose of this chapter is to motivate the approach taken in AMPS and provide a broad context in which to understand the contributions made in this thesis.

Chapter 3 introduces notation and surveys a broad class of dynamic systems, explaining how they evolve over time and defining optimal control. Under consideration are systems that change in continuous time, either in discrete stages or continuously. The chapter concludes with a brief survey of solution methods that compute optimal reactive plans.

Chapter 4 discusses generalisation, one of the primary challenges in reinforcement learning. This chapter provides a comprehensive survey of historical and state-of-the-art approaches, including abstraction, local approximation, and parametric approximation.

Chapter 5 explains modelling in AMPS. Modelling involves partitioning the state and action spaces and estimating the transition probabilities and reward functions. Modelling is defined abstractly so that different underlying data structures and algorithms can be used depending on the requirements of the application.

Chapter 6 describes ways to separate regions of the state and action spaces. This chapter discusses separation for decision graph and nearest neighbour approaches, along with issues arising when interleaving planning with separation.

Chapter 7 explains how to simplify both the dynamics of the model and the representation of the model. Since the modelling process is constantly updating and adapting the model online, the model is likely to become more complex than necessary over time. This chapter presents algorithms for managing complexity.

Chapter 8 discusses planning in greater depth with the assumption that the state and action spaces are small and the dynamics follow a semi-Markov decision process. This chapter compares a set of efficient algorithms for prioritised

planning in problems where the model is initially unknown to the agent and must be estimated from experience.

Chapter 9 explains how the concepts and algorithms for modelling and planning discussed in the previous chapters can be integrated into a single system. This chapter outlines the implementation of AMPS and discusses the interaction of its components.

Chapter 10 evaluates AMPS as a system for learning intelligent behaviour through interaction, beginning with a demonstration of AMPS working on a variety of problems. This chapter then closely examines the behaviour of the various components of AMPS and compares the system with alternative approaches.

Chapter 11 concludes the thesis with a summary and ideas for further research.

Appendix A contains proofs of the theorems referenced in the body of this thesis. Appendix B explains how to randomly generate semi-Markov decision processes.

The full text of this thesis along with supplementary material, source code, and errata is available online at `http://mykel.kochenderfer.com/thesis`.

Chapter 2

Approaches

This chapter provides a broad overview of various approaches to constructing intelligent behaviour, including programming, supervised learning, fitness optimisation, and reinforcement learning. The purpose of this chapter is to provide the context and motivation for the adaptive modelling and planning approach. Additionally, this section introduces a number of concepts that appear later in this thesis.

2.1 Introduction

Much of artificial intelligence is concerned with constructing autonomous agents. This chapter discusses some of the major approaches taken within the community, but it does not provide an overview of the entire field.[1] The main approaches considered in this chapter are as follows:

- **Programming:** The agent follows a static policy explicitly programmed by a designer. (Section 2.2)
- **Supervised Learning:** The agent tries to mimic the behaviour of a designer, attempting to generalise from taught instances. (Section 2.3)
- **Fitness Optimisation:** The agent searches for a policy that maximises some fitness measure provided by a designer. (Section 2.4)
- **Planning:** The agent reasons about the effects of its actions to decide upon a course of action that will accomplish some goal. (Section 2.5)

[1] There are many excellent introductory texts that survey the field of artificial intelligence, including those by Russell and Norvig (2003) and Nilsson (1998).

- **Reinforcement Learning:** The agent learns to maximise its expected return through interaction. (Section 2.6)

Different approaches have different advantages and disadvantages, and some approaches are suitable for some applications but not others. They make different assumptions about the problem to be solved and what counts as "intelligent" behaviour. The choice of approach, or combination of approaches, depends on the availability of expert designers and their knowledge of the problem domain. However, for applications where the agent does not possess an accurate model of the world and must interact with its environment in real time without relying upon a domain expert, typically the best approach to take is model-based reinforcement learning, which is the one this thesis adopts.

2.2 Programming

One way to produce a seemingly intelligent agent is by having a domain expert specify the behaviour of the agent explicitly. Most applications require that the agent respond to stimuli, and so a behaviour specification requires an explicit representation of a *policy*, a mapping from states as perceived through a set of sensors to actions or sequences of actions. This section considers approaches where the policy is designed by an expert.

One of the challenges in designing behaviour is deciding how to represent the policy. A table mapping states to actions is one way to specify behaviour, but large state spaces make this approach impractical. It is common to partition the state space using a *decision graph* and specify actions for entire regions of the state space. A decision graph is a directed acyclic graph with tests associated with the internal nodes and actions associated with the external nodes. Exactly one of the internal nodes is designated as the root. When the agent encounters a new state, it begins by evaluating the test associated with the root node. The result of the test determines which node to evaluate next. The procedure continues until it reaches an external node and returns the associated action for the agent to execute. Figure 2.1 shows an example of a decision graph.

Another compact way of representing hand-crafted policies is with *decision lists*. Decision lists consist of an ordered set of decision rules, which are of the form $c \rightarrow a$ where c is a condition (a binary test) and a is an action or a call to another decision list. When the agent encounters a new state, it executes

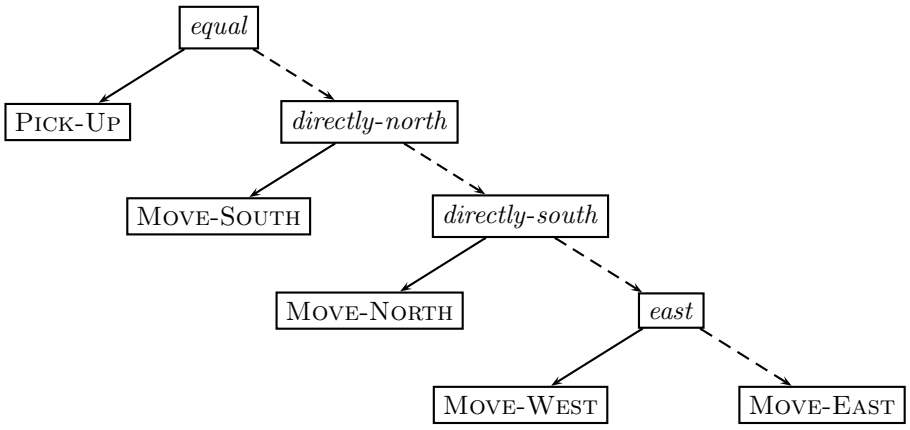


Figure 2.1: A decision graph. This decision graph encodes a policy for navigating to passengers and picking them up in the Taxi World domain. The internal nodes compare the position of the taxi with respect to the passenger. Solid lines are followed when the tests evaluate to true and dashed lines are followed when the tests evaluate to false. The top node is the root.

the action belonging to the first rule in the sequence whose condition holds. The decision list in Figure 2.2 encodes the same policy as the one encoded by the decision graph in Figure 2.1. Calls to other decision lists may involve the passing of parameters, thereby enabling recursion and hierarchical organisation of behaviour (Nilsson, 1994, 2001).

For some problems, the behaviour of the agent is better controlled by a digital computer program written in some general-purpose programming language. For example, the ELIZA conversation agent (Weizenbaum, 1966) from the early days of artificial intelligence computes its response to human input by performing

$$\begin{aligned} \textsc{Equal} &\rightarrow \textsc{Pick-Up} \\ \textsc{Directly-North} &\rightarrow \textsc{Move-South} \\ \textsc{Directly-South} &\rightarrow \textsc{Move-North} \\ \textsc{East} &\rightarrow \textsc{Move-West} \\ \top &\rightarrow \textsc{Move-East} \end{aligned}$$

Figure 2.2: A decision list. This decision list encodes the same policy as the one encoded by the decision graph in Figure 2.1.

operations on lists stored in computer memory. It would be impractical to create a decision graph or decision list that exhibits the same flexibility as the ELIZA program. Likewise, it would be difficult to build a robot that uses a video camera for navigation without specialised computer algorithms to process the sensory data.[2]

For problems that are well-defined and understood, the programming approach is most appropriate. If a domain expert knows exactly how to solve a problem, then there is little reason to have the agent find its own solution. The agent will behave exactly as specified so long as the assumptions made about the environment always hold.

2.3 Supervised Learning

This section introduces *supervised learning* as a way of achieving intelligent behaviour.[3] In supervised learning, the agent learns its policy directly from a teacher, presumably a teacher who is good at achieving some task. Since the agent attempts to learn the same behaviour exhibited by the teacher, this process is sometimes called *behavioural cloning*. During the process of behavioural cloning, the agent revises its representation of the policy to conform to the mapping exhibited by the teacher.

Donaldson (1960) describes some of the earliest work on behavioural cloning for the *pole-and-cart* task. The pole-and-cart task involves balancing a pole mounted on a wheeled cart by applying forward and reverse forces on the cart. The cart is situated on a one-dimensional track of fixed length. The agent senses the state of the world s, represented as a vector of real values. The system described by Donaldson attempts to learn a parameter vector θ such that $s \cdot \theta$ matches the control of a teacher during a training phase. Learning the parameter θ involves using gradient descent to reduce the error between what the system predicts and what the teacher specifies. Widrow and Hoff (1960) describe a similar adaptive linear approach for learning from a teacher.

Neural networks can represent policies that are more complex than the linear policies explored by Donaldson (1960) and Widrow and Hoff (1960). There are

[2]Specialised computer programs can be combined with decision graphs and decision lists. The tests might be the output of some algorithmic process and the actions might be calls to programs that control lower-level behaviour.

[3]Duda et al. (2000) provide an excellent survey of various supervised learning methods.

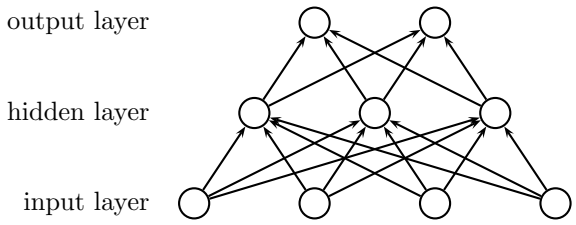


Figure 2.3: A multi-layer, feed-forward neural network. In this particular network, the input layer has four units, the hidden layer has three units, and the output layer has two units. In general, a neural network may consist of multiple hidden layers.

many kinds of neural networks,[4] but the kind typically found in supervised learning applications involve multi-layer, feed-forward networks consisting of an input layer, one or more hidden layers, and an output layer (Figure 2.3). The *units* and weighted edges in the network form a directed acyclic graph. The weight of the edge from u' to u is denoted $\theta(u', u)$, the set of predecessors of a unit is denoted $\text{pred}(u)$, and the activation of an input unit u is denoted $x(u)$. If the activations of the input units are set to the values in a factored state representation, it is possible to compute the activation of the other units. The activation at a unit u is

$$x(u) = f\left(\sum_{u' \in \text{pred}(u)} \theta(u', u) x(u')\right),$$

where f is some activation function. A gradient-descent learning algorithm such as back propagation (Rumelhart et al., 1986) tunes the weights to minimise the error between the predicted action and the actual action of the teacher. The ALVINN system (Pomerleau, 1993) uses the back-propagation algorithm to learn to steer real automobiles long distances on public highways at speeds up to 70 miles per hour (30 m/s).

Decision trees, which are decision graphs where every non-root node has exactly one parent, can also represent learned policies. Decision tree induction systems (e.g., Quinlan, 1993) can learn, for example, to control simulated aircraft, from Cessnas (Bratko and Urbančič, 1997) to F-16 combat planes (Michie and Camacho, 1994). After the teacher trains the agent for a period of time, the agent compiles its experience into a decision tree. Decision tree induction involves in-

[4]Haykin (1999) provides a comprehensive introduction to different kinds of neural networks, including feed-forward networks.

crementally splitting the training instances according to some heuristic, typically *information gain* (Shannon, 1948). Utgoff et al. (1997) explore online methods for inducing decision trees, allowing training instances to be added incrementally to an existing decision tree.

Khardon (1999) demonstrates an algorithm based on that of Rivest (1987) that creates decision lists from training experience. The algorithm begins with an empty decision list and adds rules over a series of iterations. At each iteration, the algorithm enumerates all of the possible rules and chooses the one with the best prediction rate. All of the training examples covered by that rule are removed from consideration, and the process continues until exhausting all the examples. Khardon demonstrates this algorithm on the Blocks World and Logistics domains.

Instance-based methods can be used to generalise training instances in behavioural cloning. Instance-based methods use the *nearest neighbour* algorithm (Cover and Hart, 1967) to make decisions. As the agent acquires training instances from a teacher, it stores the state-action pairs in memory. When the agent is asked to make decisions on its own, it looks for the training instance whose state is most similar to the current state according to some metric. Metoyer and Hodgins (2000) demonstrate a system based on this approach that learns how to animate defensive behaviour in American football from data collected from human players.

Although supervised learning has been successfully applied to a variety of difficult control problems, there are many drawbacks inherent in this approach. Behavioural cloning requires the presence of a competent teacher, but for some problems this is not possible. If the agent finds itself in a situation that is not related to one of its training instances, then it is not likely to be successful. Typically, many teaching episodes are necessary to sufficiently cover the state space.

2.4 Fitness Optimisation

If there exists a metric for measuring the quality of behaviour exhibited by a policy, then one may use a *fitness optimisation* approach that searches the space of possible policies for one that exhibits the greatest fitness. *Fitness* is a measure of how well a policy performs on a problem.

Fitness is usually measured experimentally rather than analytically. Measur-

ing the fitness of a football-playing agent might involve counting the number of goals the agent scores over the course of fifty games. Fitness optimisation often requires measuring the fitness of many different policies, so it is often desirable to make fitness evaluation as efficient as possible.

The remainder of this section discusses local-search and evolutionary techniques for fitness optimisation. They differ in how they search the space of possible policies. Although these approaches can produce successful controllers for a variety of problems, they generally require a tremendous amount of computation before finding a suitable policy. Once a policy is chosen, the agent is not able to adapt its policy quickly as it acquires more experience.

2.4.1 Local-Search Approaches

Searching through the entire policy space for a suitable policy is not practical because the space is typically extremely large or infinite. An alternative to exhaustive search is *local search*, also known as *hill climbing* or *gradient ascent*. Local search operates with the assumption that the fitness of a policy indicates how close the policy is to a global optimum and searches through policy space in the direction of steepest increase in fitness.

Some local search techniques directly estimate the gradient of the fitness for a particular policy. Such *policy-gradient* methods have been used to solve reinforcement learning problems with complete and partial observability (Cao, 2005; Baxter and Bartlett, 2001).[5] Because policy-gradient methods typically learn extremely slowly (Williams, 1992), Sutton et al. (2000) suggest a way to incorporate the value-function estimation techniques of Section 4.4 to speed convergence.

Other local search techniques evaluate the *neighbourhood* (or a finite sampling of the neighbourhood) of some initial policy. A neighbourhood of a policy is a set of similar policies as determined by representation. For example, the neighbourhood of a policy encoded by a neural network might be a set of policies encoded by neural networks with minor changes in their weights. The search algorithm then moves to the neighbour with the highest fitness, evaluates its neighbourhood, and the process continues.

Local search is susceptible to *local maxima* and *plateaus* in the fitness landscape. One way to handle local maxima is to introduce randomness into the

[5]Section 3.5 discusses observability in reinforcement learning problems.

search. Instead of always moving to the neighbour with greatest fitness, the search might consider neighbours with lower fitness according to some randomised exploration strategy. As the search progresses, the randomness in the exploration decreases according to some schedule. One such process is called *simulated annealing* because of its analogy with annealing in metallurgy (Kirkpatrick et al., 1983).

Another approach for handling local maxima and plateaus is *tabu search*, as surveyed by Glover and Laguna (1997). Tabu search involves maintaining a *tabu list* containing recently visited points in policy space. The search algorithm avoids neighbours in the tabu list.

2.4.2 Evolutionary Approaches

Another class of fitness optimisation methods derives inspiration from biological evolution. Holland (1975) presents *genetic algorithms* as a means of solving general optimisation problems.[6] Genetic algorithms evolve populations of (typically binary) strings, starting with an initial random population. The strings recombine through genetic crossover and mutation at a rate proportional to their measured fitness to produce a new generation. The process of evolution continues until arriving at a satisfactory solution.

Koza (1992) explores the evolution of tree structures using what he calls *genetic programming*. Tree structures allow a more flexible representation for policies than strings. Trees consist of symbols selected from predefined sets of terminals and non-terminals. Crossover works by swapping subtrees, and mutation works by randomly modifying subtrees. Kochenderfer (2003) uses genetic programming to evolve policies encoded as decision lists for solving general block-stacking problems.

Genetic algorithms and genetic programming may be combined with other methods, including local search. For example, a genetic algorithm might evolve a satisfactory policy and then use local search to further improve the policy. Such an approach is called *genetic local search* or *memetic algorithms* and has recently been the subject of much research (Hart et al., 2005).

Typically, evolutionary approaches involve evaluating the fitness of large populations. Fortunately, evaluation may be done in parallel and the computational

[6] Mitchell (1996) surveys recent research in genetic algorithms.

load may be distributed across multiple processors. Because it is generally impractical to evaluate the performance of large populations of physically embodied agents, fitness measures are usually done in simulation. Once a suitable policy has been evolved, it is transferred to a real robot. Walker et al. (2003) surveys the application of evolutionary approaches to physical robots.

2.5 Planning

The previous three sections introduced programming, supervised learning, and fitness optimisation as ways to construct intelligent behaviour. None of these approaches involve the agent knowing anything about the dynamics of the world. This section introduces *planning* as a way for the agent to reason about how the world changes in response to its course of action.

Planning may refer to algorithms that produce plans for *open-loop* or *closed-loop* behaviour. Open-loop plans are simply sequences of actions that aim to achieve some goal. In non-deterministic worlds, a static sequence of actions may not always have the predicted effect. If the agent perceives that the plan is not proceeding as expected, it may replan from the current state.

Open-loop planning may be thought of as a search process. Given some initial state, the planning process applies a series of actions to transform the current state with the objective of finding a minimal cost path to some goal. This process may be implemented using *breadth-first search*, *depth-first search*, *iterative-deepening search*, or some form of *heuristic search* (Hart et al., 1968, 1972).

The STRIPS system developed by Fikes and Nilsson (1971) applies automated theorem proving to solving open-loop planning problems. STRIPS uses first-order predicate calculus to specify the preconditions and the effects of various actions. Planning proceeds backwards from the desired goal represented as a logical formula. Nilsson (1984) used STRIPS in Shakey, the first robot capable of planning and reasoning about action. The STRIPS approach inspired the development of many other automated planning methods, as surveyed by Ghallab et al. (2004).

Closed-loop planning research originates in the work by Bellman (1957) where he introduces *dynamic programming* as a means to efficiently compute optimal policies for non-deterministic environments. He assumes that the world behaves

as a *Markov decision process* (MDP) where the next state depends probabilistically on the current state and the action taken by the agent. His work can be generalised to environments where state transitions occur in continuous time (Chapters 3 and 8).

Planning requires a model of the dynamics of the underlying system. The dynamics may be specified by a human, or the agent may learn the system model through experience (Chapter 5).

2.6 Reinforcement Learning

The objective of *reinforcement learning*[7] is to learn a policy that maximises expected discounted return. Reinforcement learning methods assume that the system dynamics follow some class of model, such as the *semi-Markov decision process* or the *controlled diffusion process* from Chapter 3. In contrast with the previous section, the actual behaviour of these systems does not have to be completely known.

Reinforcement learning techniques are flexible and can solve a wide variety of problems. Some problems, such as Taxi World and Corner World, involve *goals of achievement* where the objective of the agent is to reach some goal state. There are different ways of framing such problems. One way is to penalise the agent at some rate until it reaches a goal. Another way is to provide positive reward at the goal state and no reward at all other states and discount the reward received at some rate. This thesis focuses on discounted problems, where the agent attempts to maximise its expected discounted return.

Some problems do not involve achieving any particular goal state but instead have some *goal of maintenance* where the agent must maximise some reinforcement signal over time. For example, McCallum (1996a) presents the "New York driving" problem where the agent must weave in and out of one-way traffic. The agent receives large negative reward for scraping by slower trucks, small negative reward for driving too slow in front of a faster truck, and small positive reward for making forward progress. Both goals of achievement and goals of maintenance reduce to maximising expected discounted return.

[7] Sutton and Barto (1998) provide a comprehensive introduction to the field of reinforcement learning, including a historical overview. Kaelbling et al. (1996) provide an excellent survey of research in the area.

The literature typically categorises reinforcement learning algorithms according to whether or not they involve learning a representation of the system dynamics. The remainder of this section provides a brief overview of model-free and model-based approaches. Chapter 3 discusses reinforcement learning methods in greater detail.

2.6.1 Model-Free Approaches

Model-free reinforcement learning does not involve learning an explicit representation of the system dynamics. Instead, model-free approaches attempt to directly learn a mapping from states and actions to their expected discounted return. This mapping is called the *action-value function*. When the agent encounters a new state, it chooses its next action according to this function. If the agent does not possess an accurate value function, it must occasionally choose non-greedy actions in order to learn an optimal policy in the limit. Deciding between exploiting what appears to be the best action and exploring other alternatives is an important issue in reinforcement learning, and the literature contains suggestions for many heuristic approaches (e.g., Thrun, 1992; Wiering, 1999).

Without a model, the agent has no concept of how the world behaves and cannot reason about the effects of its actions. It only knows approximately how much discounted return to expect when taking a particular action. If the goals of the task or the reward structure change, then the agent must relearn the value function from direct interaction with the world instead of using a learned model as a basis to replan. Learning a satisfactory value function generally requires extensive interaction with the world. However, model-free approaches have been successful in a wide variety of domains from vision-based robotics (Asada et al., 1996) to game playing (Baxter et al., 2000; Tesauro, 2002). There are two classes of model-free algorithms, one based on *Monte Carlo estimation* and the other based on *temporal-difference learning*. Sections 3.4.2 and 3.4.3 discuss these solution methods.

2.6.2 Model-Based Approaches

Kumar (1985) divides *model-based* approaches into two categories, one using a *Bayesian* formulation and the other using a *non-Bayesian* formulation. The complexity of the Bayesian approach limits its applicability to relatively simple prob-

lems. Hence, this thesis focuses on a non-Bayesian approach to model-based reinforcement learning.

The Bayesian formulation of the reinforcement learning problem assumes the model of the system dynamics depends exclusively on a parameter θ in the set Θ. The agent updates its probability distribution over Θ using Bayes' rule as it accumulates experience, starting with some prior probability distribution over Θ. The agent chooses the action that maximises its expected discounted return with respect to its probability distribution over Θ. To model the uncertainty of the transition probabilities, it is common to use a Dirichlet prior or a hierarchical prior that incorporates the assumption that connectivity of the transition model is sparse (Friedman and Singer, 1999). The prior over rewards and transition duration may be modelled by normal and gamma distributions, which are conjugate priors for the normal and exponential distributions. Analytic solutions only exist for very simple problems, but solutions for more complex problems may be approximated using Monte Carlo techniques (Dearden et al., 1999; Strens, 2000; Duff, 2002).

The non-Bayesian approach does not maintain a probability distribution over Θ, but instead it updates a single estimate $\hat{\theta}$ of the true parameter θ according to experience. So long as the agent follows a suitable exploration policy and uses a suitable estimation method, such as maximum-likelihood estimation (Fisher, 1922), $\hat{\theta}$ will converge to θ with sufficient experience.[8] At each update of $\hat{\theta}$, the agent applies dynamic programming to revise its value function and policy. This approach leads to an optimal policy in the limit.

In general, model-based approaches learn higher quality policies in less time and with less data than model-free approaches (Moore and Atkeson, 1993; Peng and Williams, 1993; Atkeson and Santamaría, 1997). These empirical results are not surprising because model-free approaches cannot perform global replanning like the model-based approaches.[9] Model-based approaches have an advantage over model-free approaches because they can use the learned model of the environment across tasks with different reward structures (cf. Mahadevan, 1992).

There appears to be some elements of model-based learning in humans. Recent research in neuropsychology indicates that humans perform some form of

[8] There are different types of convergence for stochastic sequences, including strong and weak convergence. Whether $\hat{\theta}$ converges strongly or weakly to θ depends on the estimation strategy. For an introduction to stochastic limit theory, see the text by Davidson (1994).

[9] See also the theoretical analysis by Kearns and Singh (1999).

model-based reinforcement learning. Yoshida and Ishii (2005), for example, use functional magnetic resonance imaging to support the hypothesis that the dorsolateral prefrontal cortex maintains and manipulates world environmental models and the anterior cingulate cortex performs action selection.

2.7 Discussion

As this chapter shows, there are many different kinds of approaches one may use when constructing an intelligent agent. Depending on the nature of the problem, some approaches are more suitable than others. This section summarises the advantages and disadvantages of the approaches considered in this chapter.

Programming the behaviour of an agent explicitly involves encoding the policy using some general-purpose programming language or a compact data structure such as a decision graph or decision list. Such an approach can be quite effective so long as the designer has a sufficiently detailed understanding of the problem and how to solve it. Unfortunately, if the world does not behave as the designer expects, the agent cannot adapt its policy to solve the problem.

In supervised learning, the agent learns by observing a teacher. The agent generalises from individual training instances using methods such as neural-network back propagation, decision tree induction, or nearest neighbour generalisation. Such an approach is useful when a designer knows how to solve a problem but does not know exactly how to program a solution. The success of supervised learning for behavioural cloning depends upon a high-quality teacher and suitable training examples. As with the programming approach, the agent is unable to adapt its behaviour in response to its experience in the world without the assistance of a teacher.

Fitness optimisation techniques are useful when the designer does not know how to solve a problem but can measure the quality of a solution. Local search and evolutionary methods enjoy many successful applications, but they can be computationally expensive since they generally require many policy evaluations before finding a suitable policy.

Planning algorithms compute policies given some model of the world dynamics. There are many different kinds of planning algorithms, some producing open-loop plans and others producing closed-loop plans. A reactive agent requires a closed-loop plan, and there are a variety of dynamic-programming algorithms

that can compute optimal, closed-loop plans efficiently. The success of a planning algorithm in practice depends on the accuracy of the model.

Reinforcement learning algorithms do not require complete knowledge of the underlying model. Instead, they adapt their policies in response to the incoming stream of experience. Model-based reinforcement learning involves constructing a model from experience and then applying a planning algorithm to compute a policy. Model-free algorithms compute policies from estimates of return without attempting to model the dynamics of the world. Model-based reinforcement learning approaches learn much more efficiently than model-free approaches and are much more suitable for an agent that must solve a problem in an unknown world with little experience. This thesis therefore focuses on model-based approaches for learning intelligent behaviour.

Before discussing model construction, it is necessary to understand the underlying system dynamics of the class of problems under consideration. The next chapter discusses both discrete and continuous event dynamic systems and conditions for optimality.

Chapter 3

Systems

This chapter introduces models of discrete and continuous event dynamic systems, presenting the fundamental concepts and notation used throughout this thesis. The first section introduces dynamic control problems. The second and third sections present the dynamics and optimality conditions for discrete and continuous systems. The fourth section discusses solution methods, and the fifth section considers issues with partial observability. The purpose of this chapter is to provide the necessary technical background for understanding the context and contributions of this thesis.

3.1 Introduction

A model of a dynamic system specifies how the state of the system evolves over time in response to the actions taken by the agent. Throughout this thesis, $\mathbb{S}$ represents the state space and $\mathbb{A}$ represents the action space. For many problems, it is useful to restrict the actions available from particular states. Let $\mathbb{A}(s) \subset \mathbb{A}$ be the set of actions available from state s. Different classes of dynamic systems make different assumptions about the set of states and the set of actions.

The instant the state changes and the agent makes a control decision is called an *event*. Dynamic systems may be categorised according to whether events occur discretely or continuously. Discrete-event systems are usually modelled by *semi-Markov decision processes* (SMDPs), and continuous-event systems are usually modelled by *controlled diffusion processes*. This chapter describes both kinds of systems in detail.

Reinforcement learning problems involve an agent that accumulates reward

while interacting with the world. The agent may receive positive or negative reward both continuously at some rate and discretely in lump sums. The objective of the agent is to find a policy that maximises the expected discounted return. Throughout this thesis, the *continuous compound discount rate* is denoted $\beta \in (0, \infty)$. Any reward received at time t is discounted by a factor $e^{-\beta t}$, thereby encouraging the agent to pursue reward more aggressively.

A *policy* specifies the behaviour of the agent. When the word policy is used in this thesis, it generally refers specifically to a *stationary deterministic policy*, which is a mapping $\pi : \mathbb{S} \to \mathbb{A}$. Stationary deterministic policies are also frequently called decision rules, reactive plans, universal plans, or strategies. Other kinds of policies include stationary stochastic policies, non-stationary deterministic policies, and non-stationary stochastic policies.

Optimality is defined using value functions. A *value function* is a mapping $V : \mathbb{S} \to \mathbb{R}$. A partial ordering may be defined over the space of value functions such that $V \leq V'$ if and only if $V(s) \leq V'(s)$ for all states s. The L^∞-norm can be defined over the space of value functions where $\|V\| \equiv \max_{s \in S} V(s)$. Addition and subtraction can be defined where $(V + V')(s) = V(s) + V'(s)$ and $(V - V')(s) = V(s) - V'(s)$. The value function V^π evaluated at a state s is the expected discounted return when starting at state s and following the policy π. The *optimal value function*, written V^*, is defined to be

$$V^* \equiv \max_{\pi \in \Pi} V^\pi,$$

where Π is the space of all stationary deterministic policies. An *optimal policy*, written π^*, is a policy that satisfies $V^* = V^{\pi^*}$.

Although *action-value functions* are not necessary to define optimality, they are useful when discussing solution methods. The action-value function Q^π evaluated at a state s and action a is the expected discounted return when taking action a from s and then following π. The *optimal action-value function* Q^* evaluated at a state s and action a is the expected discounted return when starting at state s, taking action a, and continuing with an optimal policy. Hence, the following equalities hold:

$$\begin{aligned} V^\pi(s) &= Q^\pi(s, \pi(s)) \\ V^*(s) &= \max_{a \in \mathbb{A}(s)} Q^*(s, a). \end{aligned}$$

This thesis focuses on the infinite-horizon discounted reward optimality criterion because it is the simplest to analyse and is typically the most useful in

practice. Another useful optimality criterion is the undiscounted finite-horizon criterion, which is useful when the lifetime of the agent is known in advance. The infinite horizon, average-reward optimality criterion does not require a discount rate and is useful in a number of domains (see Mahadevan, 1996). Kaelbling et al. (1996, Section 1.2) and Littman (1996, Section 1.3.1) discuss these other optimality criteria in greater detail.

3.2 Discrete Dynamic Systems

In discrete-event dynamic systems, the state changes in response to control decisions made at discrete decision stages. In general, $\mathbb{S}$ and $\mathbb{A}$ are assumed finite and some amount of time is required to transition between states. These systems are known as semi-Markov decision processes (SMDPs). SMDPs have been considered in various texts including those of Howard (1971), White (1993, Chapter 5), and Puterman (1994, Chapter 11).

Before discussing the system dynamics of SMDPs, it is important to note that most of the literature on discrete-event dynamic systems focus on *Markov decision processes*. There are two ways to view MDPs in relation to SMDPs. One way is to view an MDP as a special type of SMDP where the duration between events is constant. Another way to view an MDP is as an SMDP where the reward is received in lump sums, discounted at a constant rate per event instead of according to time. There has been some work on generalising methods developed for MDPs so that they apply to SMDPs (e.g., Bradtke and Duff, 1995). In many places, this thesis generalises work by others to conform to the SMDP model formulation in this section.

3.2.1 Dynamics

If an agent takes an action a in some state s, it will transition to a new state s' selected from a fixed probability distribution depending only on s and a. The duration of time spent in this transition and the reward received is also selected from fixed probability distributions. In particular:

- $P(s' \mid s, a)$ is the probability that taking action a in state s will result in a transition to state s'.

- $P_t(t \mid s, a, s')$ is the probability that the transition from s to s' by action a completes within time t. It is assumed that $P_t(0 \mid s, a, s') < 1$.

- $P_r(r \mid s, a, s')$ is the probability that the agent receives a lump-sum reward less than or equal to r while transitioning from s to s' by action a.

- $P_\rho(\rho \mid s, a, s')$ is the probability that the agent receives reward at a rate less than or equal to ρ while transitioning from s to s' by action a.

In addition, $r(s, a, s')$ and $\rho(s, a, s')$ represent the expected lump-sum reward and expected reward rate respectively when transitioning from state s to state s' by action a. The literature often assumes that the cumulative probability distribution functions do not depend on the state to which the agent transitions, e.g. $P_t(t \mid s, a, s') = P_t(t \mid s, a)$. However, in some problems it is important that the probability distributions depend upon the new state s'.

The experience of the agent may be written down as a sequence,

$$(s_1, t_1, a_1, r_1, \rho_1, s_2, t_2, a_2, r_2, \rho_2, \ldots),$$

where s_k is the state, t_k is the time, and a_k is the action of the kth decision. The resulting lump-sum reward and reward rate are given by r_k and ρ_k respectively. The *discounted reward* during the kth transition is defined to be

$$\begin{aligned} R_k &= e^{-\beta(t_{k+1}-t_k)} r_k + \int_0^{t_{k+1}-t_k} e^{-\beta t} \rho_k dt \\ &= e^{-\beta(t_{k+1}-t_k)} r_k + \frac{1 - e^{-\beta(t_{k+1}-t_k)}}{\beta} \rho_k. \end{aligned} \tag{3.1}$$

3.2.2 Optimality

When discussing optimal control of SMDPs, it is useful to make the following definitions:

$$\gamma(s, a, s') \equiv \int_0^\infty e^{-\beta t} dP_t(t \mid s, a, s') \tag{3.2}$$

$$\lambda(s, a, s') \equiv \int_0^\infty \int_0^t e^{-\beta t'} dt' dP_t(t \mid s, a, s') \tag{3.3}$$

$$R(s, a) \equiv \sum_{s' \in \mathbb{S}} P(s' \mid s, a) \left[\gamma(s, a, s') r(s, a, s') + \lambda(s, a, s') \rho(s, a, s')\right] \tag{3.4}$$

$$\alpha \equiv \max_{s \in \mathbb{S}} \max_{a \in \mathbb{A}(s)} \sum_{s' \in \mathbb{S}} P(s' \mid s, a) \gamma(s, a, s'). \tag{3.5}$$

Various proofs of convergence use the value $\alpha \in (0,1)$, and the functions γ, λ, and R appear throughout this thesis. The interpretation of these functions are as follows:

- $\gamma(s,a,s')$ is the discounted value of unit lump-sum reward received after transitioning from s to s' by action a.
- $\lambda(s,a,s')$ is the expected cumulative discounted value of reward received at a constant unit rate while transitioning from s to s' by action a.
- $R(s,a)$ is the total expected discounted reward when starting in s and executing action a until transitioning to some other state.

Observe that the formula for $\lambda(s,a,s')$ simplifies:

$$\begin{aligned}\lambda(s,a,s') &= \int_0^\infty \int_0^t e^{-\beta t'} dt' dP_t(t \mid s,a,s') \\ &= \int_0^\infty \frac{1-e^{-\beta t}}{\beta} dP_t(t \mid s,a,s') \\ &= \left(1-\gamma(s,a,s')\right)/\beta. \end{aligned} \tag{3.6}$$

The expected discounted return when starting at state s and following policy π is given by

$$V^\pi(s) \equiv E\left\{\sum_{k=1}^\infty \left[e^{-\beta t_{k+1}} r_k + \int_{t_k}^{t_{k+1}} e^{-\beta t}\rho_k dt\right] \mid s_1 = s, a_k = \pi(s_k)\right\}.$$

In order to calculate V^π, it is useful to define the mapping B_π from value functions to value functions such that

$$B_\pi V(s) \equiv R(s,\pi(s)) + \sum_{s' \in \mathbb{S}} P(s' \mid s,\pi(s))\gamma(s,\pi(s),s')V(s').$$

If V_0 is a value function that maps all states to 0, then $B_\pi V_0$ is the expected discounted return after taking one action according to the policy π. The expected discounted return when following π for k decisions is given by $B_\pi^k V_0$. Hence,

$$V^\pi = \lim_{k\to\infty} B_\pi^k V_0. \tag{3.7}$$

The optimal value function may be computed in a similar way. Define B, known as the *Bellman update operator*, to be a mapping such that

$$BV(s) \equiv \max_{a \in \mathbb{A}(s)} \left[R(s,a) + \sum_{s' \in \mathbb{S}} P(s' \mid s,a)\gamma(s,a,s')V(s')\right]. \tag{3.8}$$

Theorem 2 (in Appendix A) states that

$$V^* = \lim_{k \to \infty} B^k V.$$

for any value function V. With V^* known, an optimal policy π^* can be constructed as follows (see Theorem 3 in Appendix A):

$$\pi^*(s) = \arg\max_{a \in \mathbb{A}(s)} R(s,a) + \sum_{s' \in \mathbb{S}} P(s' \mid s,a)\gamma(s,a,s')V^*(s'). \tag{3.9}$$

3.3 Continuous Dynamic Systems

In continuous dynamic systems, the state changes continuously in response to continuous control. The state space $\mathbb{S}$ is assumed to be a closed subset of $\mathbb{R}^n$ with boundary $\partial\mathbb{S}$, and the action space $\mathbb{A}$ is assumed to be a compact subset of Euclidean space. This section specifies how stochastic differential equations can model the evolution of continuous dynamic systems and presents the Hamilton-Jacobi-Bellman equation as a condition for optimal control. The class of continuous dynamic systems is covered in greater depth by Øksendal (2003), Kushner and Dupuis (2001), and Kloeden and Platen (1999).

3.3.1 Dynamics

At time t, let $s(t)$ denote the state of the world and $a(t)$ denote the action taken by the agent. The evolution of the system is determined by the stochastic differential equation

$$ds(t) = f(s(t), a(t))dt + \sigma(s(t), a(t))dw, \tag{3.10}$$

where $f : \mathbb{S} \times \mathbb{A} \to \mathbb{R}^n$, $\sigma : \mathbb{S} \times \mathbb{A} \to \mathbb{R}^{n \times m}$, and w is the m-dimensional Wiener process (or Brownian motion). The function f specifies the "local drift" vector and σ specifies the "diffusion matrix." The state changes according to this stochastic differential equation until the trajectory exits the state space through the boundary $\partial\mathbb{S}$. Such a system is known as a controlled diffusion process (Borkar, 1989). More complex formulations include jump-diffusion processes that allow discontinuous movements in the form of discrete "jumps" (Kushner and Dupuis, 2001).

The agent receives reward at a rate $\rho(s,a)$ for taking action a in state s. If the agent exits $\mathbb{S}$ from state $s \in \partial\mathbb{S}$ by action a, then the agent receives a lump-sum reward of $r(s,a)$ and the episode terminates.

3.3.2 Optimality

The expected discounted return when starting at state $s(t)$ and following policy π is given by

$$V^{\pi}(s(t)) \equiv E\left\{\int_t^{\tau} e^{-\beta t'} r(s(t'), \pi(s(t')))dt' + e^{-\beta\tau} r(s(\tau), \pi(\tau))\right\},$$

where τ is the exit time from $\mathbb{S}$. If the trajectory never exits $\mathbb{S}$, then $\tau = \infty$.

Since the episode terminates as soon as the agent encounters a state on the boundary, the optimal value function $V^*(s)$ is equal to $\max_{a\in\mathbb{A}(s)} r(s,a)$ whenever $s \in \partial\mathbb{S}$. An optimal policy on the boundary is given by $\pi^*(s) = \arg\max_{a\in\mathbb{A}(s)} r(s,a)$.

Computing the optimal value function for states within the boundary is more complicated and requires results from stochastic calculus. Øksendal (2003, Chapter 11) and Kloeden and Platen (1999, Section 6.5) provide derivations of the *Hamilton-Jacobi-Bellman equation*, which states that the following equality holds:

$$\beta V^*(s) = \max_{a\in\mathbb{A}(s)} \left[\rho(s,a) + \frac{\partial V^*}{\partial s} f(s,a) + \frac{1}{2}\sum_{i,j=1}^{n} \frac{\partial^2 V^*}{\partial s_i \partial s_j}\phi_{ij}(s,a)\right], \tag{3.11}$$

where $\phi(s,a) \equiv \sigma(s,a)\sigma(s,a)^{\top}$. An optimal control policy for states within the boundary is given by

$$\pi^*(s) = \arg\max_{a\in\mathbb{A}(s)} \left[\rho(s,a) + \frac{\partial V^*}{\partial s} f(s,a) + \frac{1}{2}\sum_{i,j=1}^{n} \frac{\partial^2 V^*}{\partial s_i \partial s_j}\phi_{ij}(s,a)\right].$$

3.4 Solution Methods

This section reviews various solution methods for discrete dynamic systems as modelled by SMDPs. Continuous dynamic systems will not be discussed in detail in this section. Such systems generally cannot be solved analytically, and their solution typically involves approximation as discrete dynamic systems (Kushner, 1990). There are three broad classes of solution methods for SMDPs: dynamic programming, temporal-difference learning, and Monte Carlo estimation. This section discusses each class in turn and reviews solution methods for deterministic systems.

3.4.1 Dynamic Programming

Value iteration and *policy iteration* are two forms of *dynamic programming* developed by Bellman (1957) and Howard (1960), with a more recent and thorough treatment by Bertsekas and Shreve (1996). These techniques compute optimal policies given some model, either known *a priori* or estimated from experience.

Value iteration involves estimating the optimal value function by starting with some value function and applying the mapping B as defined in Equation 3.8 repeatedly until convergence. The optimal policy is computed from the optimal value function as shown in Equation 3.9. Further discussion of value iteration is reserved for Chapter 8.

Policy iteration begins with some policy π and computes V^π, either iteratively based on Equation 3.7 or using some system for solving linear equations such as Gaussian elimination.[1] The policy-improvement step involves updating π according to the update rule

$$\pi(s) \leftarrow \arg\max_{a \in \mathbb{A}(s)} \left[R(s,a) + \sum_{s' \in \mathbb{S}} P(s' \mid s,a)\gamma(s,a,s')V^\pi(s') \right].$$

The process of policy evaluation and improvement continues until the quality of the policy can no longer be increased.

3.4.2 Monte Carlo Estimation

In contrast with dynamic programming, *Monte Carlo* methods are model-free approaches to reinforcement learning. They use measurements of the discounted return received by the agent through a series of actual simulations. The agent can estimate $Q^\pi(s,a)$ by averaging of a series of measurements of discounted return when starting in state s, taking action a, and following π. Through a policy-improvement approach similar to policy iteration, the agent can calculate an optimal policy with sufficient experience. Some Monte Carlo algorithms estimate the optimal action-value function directly instead of using a policy improvement approach. A variety of Monte Carlo reinforcement learning algorithms are discussed by Sutton and Barto (1998, Chapter 5).

[1] The books by Golub and Van Loan (1996) and Watkins (2002) contain methods for solving systems of linear equations.

3.4.3 Temporal-Difference Learning

Like Monte Carlo estimation, *temporal-difference* learning does not require a model. However, instead of sampling discounted returns, temporal-difference learning uses a *bootstrapping* technique where the estimate of the value function at the current state is used to update the estimate of value of the previous state.

There are many different algorithms that use the temporal differencing approach (see Sutton and Barto, 1998, Chapter 6). One of the most popular methods is *Q-learning* (Watkins, 1989). This approach attempts to learn the optimal action-value function directly. After experiencing a transition of duration t from s to s' by a, the agent updates its estimate Q according to the rule[2]

$$Q(s,a) \overset{\eta}{\leftarrow} R + e^{-\beta t} \max_{a \in \mathbb{A}} Q(s',a),$$

where the value R is the discounted reward for the transition. The *temporal-difference error* is the relative increase of Q due to an update. The estimated Q will strongly converge to Q^* with a suitable exploration strategy (Watkins and Dayan, 1992; Tsitsiklis, 1994).

Sutton (1988) explores a way to combine temporal differencing with the Monte Carlo approach. His algorithm, called TD(λ), has a parameter λ that balances temporal-difference learning with Monte Carlo estimation. Sutton and Barto (1998, Chapter 7) explain in detail how to use *eligibility traces* to combine temporal-difference learning with Monte Carlo estimation.

3.4.4 Deterministic Solutions

Many important kinds of dynamic systems behave deterministically in response to the actions taken by the agent. Some deterministic systems can be framed as *shortest-path* problems. A solution to a shortest-path problem is the policy that maximises the expected *undiscounted* return. In general, shortest-path problems assume that there exists at least one terminal state and that the optimal policy will lead to a terminal state.

Dijkstra's algorithm (Dijkstra, 1959) can solve shortest-path problems with positive costs in $O(|\mathbb{A}| + |\mathbb{S}| \log |\mathbb{S}|)$ time when using a Fibonacci heap (Fredman and Tarjan, 1987). Since shortest-path problems consider undiscounted reward, the cost of taking an action from a state is simply $-(r + \rho t)$, where r is the

[2]The notation $X \overset{\eta}{\leftarrow} Y$ is shorthand for $X \leftarrow (1-\eta)X + \eta Y$.

expected lump-sum reward, ρ is the expected reward rate, and t is the expected duration of the transition.[3] To be solvable by Dijkstra's algorithm, this cost must be non-negative. The Bellman-Ford algorithm (Bellman, 1958; Ford and Fulkerson, 1962) runs in $O(|\mathbb{S} \times \mathbb{A}|)$ time and can solve shortest path problems even with positive and negative costs.

3.5 Observability

This section discusses *observability* where the agent makes observations that are probabilistically related to the underlying state of the world. In particular, if the agent starts in state s, takes action a, and transitions to state s', the agent will observe o with probability $P_o(o \mid s, a, s')$. If the agent has access to these observations instead of the actual state, the system might appear to follow a *non-Markov* process that cannot be modelled by an SMDP. Such a process is called a *partially observable semi-Markov decision process* (POSMDP).

Before discussing POSMDPs whose observations are randomly generated from arbitrary probability distributions, there are two special cases worth noting, *fully observable* and *unobservable* processes. If the world emits observations according to a bijection that maps the current state to an observation, then the process is fully observable. Fully observable systems appear to follow an SMDP, allowing the application of the solution methods from the previous section. If the world produces observations independent of the underlying state, then the process is unobservable.

Solution methods for POSMDPs typically maintain a *belief state* as the agent acquires experience.[4] A belief state is a probability distribution over the state space, and the *belief space* is composed of all possible belief states. When the agent commences its interaction with the world, it is uncertain of its initial location. This uncertainty is modelled by a probability distribution b, which is the initial belief state.

The objective of the agent is to maximise its expected discounted reward given its past history and initial belief state. As the agent interacts with the world, it

[3] Although shortest-path problems assume deterministic state transitions, there is no restriction prohibiting transition durations, lump-sum rewards, and reward rates from following arbitrary probability distributions. Calculating an optimal policy only requires that the means of these distributions be known.

[4] Alternatively one might use a predictive state representation approach (Littman et al., 2002; James, 2005).

can update its belief state based on its previous belief state b and the most recent action a, transition duration t, and observation o. The application of Bayes' rule and the law of total probability lead to the following update rule:

$$b(s') \leftarrow \kappa P_o(o \mid s, a, s') \left(\sum_{s \in \mathbb{S}} \frac{dP_t(t \mid s, a, s')}{dt} b(s) \right) \sum_{s \in \mathbb{S}} P(s' \mid s, a) b(s),$$

where κ is a normalisation constant such that $\sum_{s \in \mathbb{S}} b(s) = 1$ for the new belief state b.

Policies for POSMDPs are mappings from belief states to actions. Finding an optimal policy of a POSMDP involves reformulating the problem as an SMDP over the continuous space of belief states. Kaelbling et al. (1998) explain how to perform this transformation in the discrete-time case and how to go about computing an optimal policy. Madani et al. (2003) show that, in general, computing an optimal policy is undecidable. There are several approximation techniques for solving MDPs defined over the belief state. Hauskrecht (2000) surveys value-function approximation techniques, and Thrun (2000) shows how to use a Monte Carlo approach to approximate solutions in the discrete-time case.

Another approach for approximating solutions to partially observable problems is by using experience history directly instead of maintaining belief states. History-based algorithms are well suited for problems where the agent does not know the parameters of the underlying system and must build a model from experience. These algorithms disambiguate the current state using past observations. In his doctoral thesis, McCallum (1995) proposes different ways of deciding which aspects of the past observations are relevant to disambiguate the current state. Whitehead and Lin (1995) discuss alternative algorithms.

Partially observed controlled diffusions are continuous dynamic systems where the underlying process is modelled by a controlled diffusion and the observations of the underlying state are generated randomly. The observation $o(t)$ at time t changes according to the stochastic differential equation

$$do(t) = f'(s(t), a(t))dt + \sigma'(s(t), a(t))dw',$$

where $f' : \mathbb{S} \times \mathbb{A} \to \mathbb{R}^{n'}$, $\sigma' : \mathbb{S} \times \mathbb{A} \to \mathbb{R}^{n' \times m'}$, and w' is the m'-dimensional Wiener process. The process w' is independent of the process w generating the change in state. Fleming and Pardoux (1982) and Sondik (1978) discuss the control of partially observed diffusions and various solution methods.

3.6 Representation

Models of discrete-event dynamic systems require the setting of many parameters. For each state-action-state triple, the model defines reward, duration, and transition probabilities, requiring $O(|\mathbb{S}|^2 |\mathbb{A}|)$ entries in a table.[5] However, it is often the case that there is some structure underlying the system dynamics allowing a more compact representation. This section discusses how dynamic Bayesian networks and probabilistic STRIPS can represent the system model.

3.6.1 Dynamic Bayesian Networks

One way to capture structure in discrete-event systems is with *dynamic Bayesian networks* (Dean and Kanazawa, 1989), which are Bayesian networks with two layers of nodes that model probabilistic change in state.[6] For such a representation to be useful, the state or the action space must use a *factored representation* where $s = (s_1, \ldots, s_n)$ and $a = (a_1, \ldots, a_m)$. Figure 3.1 shows a dynamic Bayesian network capturing the probabilistic dependency of the transition duration, reward, and state components at event $k+1$ on the state and action components at event k. An arrow from one node to another indicates that the value of the destination node depends probabilistically on the value of the source node. Associated with the destination nodes are representations of the conditional probabilities.

Not only do factored representations lend themselves to compact representations, but they also allow the application of efficient algorithms for computing an optimal or approximately optimal policy. Boutilier et al. (2000) describe variations of the standard value iteration and policy iteration algorithms that leverage the structure encoded in a dynamic Bayesian network. Their work assumes that the actions are not factored, the state variables take on binary values, and the system follows an MDP. They use decision trees to represent the conditional probabilities. Hoey et al. (2000) improve the efficiency of structured value iteration by using *algebraic decision diagrams* (Bahar et al., 1997) instead of decision trees. Algebraic decision diagrams are decision graphs with real values associated with the external nodes. Kim and Dean (2002) discuss how to use *model-minimisation*

[5] For most real-world problems, these matrices are sparse, i.e. they contain mostly zeros, because most transitions have zero probability. Instead of using matrices, it is typically more efficient to only represent the parameters of the non-zero transitions using arrays linked in memory.

[6] For an introduction to Bayesian networks, see the books by Pearl (1988) and Neapolitan (2004).

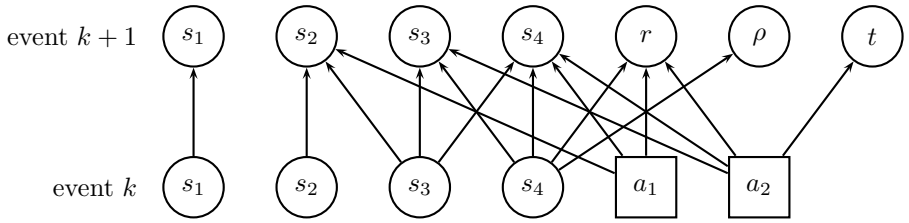


Figure 3.1: A dynamic Bayesian network. The transition duration, reward, and state variables at event $k+1$ depends probabilistically on the state and action components at event k. The arrows indicate the direction of probabilistic influence from one variable to another.

$$\textsc{Pickup}(X) \quad : \quad on(X,Y) \wedge clear(X) \wedge hand\text{-}empty$$
$$\rightarrow \begin{cases} 0.8 & \neg\, on(X,Y) \wedge inhand(X) \wedge \neg\, hand\text{-}empty \wedge clear(Y) \\ 0.1 & \neg\, on(X,Y) \wedge on(X, table) \wedge clear(Y) \\ 0.1 & \text{no change} \end{cases}$$

Figure 3.2: A probabilistic STRIPS rule. This example is based on the noisy Blocks World domain studied by Zettlemoyer et al. (2005).

techniques to reduce MDPs with factored states and actions to equivalent, but smaller, MDPs that can be solved with traditional techniques. Using dynamic Bayesian networks to represent reinforcement learning problems continues to be an active area of research.

3.6.2 Probabilistic STRIPS

Another way to model large MDPs is with a generalisation of STRIPS rules (Fikes and Nilsson, 1971) where actions may have probabilistic effects (Hanks and McDermott, 1994; Younes et al., 2005). In *probabilistic STRIPS*, a set of rules define the dynamics of the world. Figure 3.2 shows a probabilistic STRIPS rule that specifies the effects of the action PICKUP in a noisy Blocks World domain.

When an agent takes an action, the system searches through the list of probabilistic STRIPS rules for one that covers the current situation. Suppose that

block a is clear and resting on top of block b and that the hand is empty. If the agent executes the action PICKUP(a), then the rule in Figure 3.2 applies, binding a to the variable X. The precondition

$$on(a, Y) \wedge clear(a) \wedge hand\text{-}empty$$

holds with b bound to Y. As specified by the rule, with probability 0.8,

$$\neg\, on(a, b) \wedge inhand(a) \wedge \neg\, hand\text{-}empty \wedge clear(b)$$

holds. The following sentence holds with probability 0.1,

$$on(a, b) \wedge on(a, table) \wedge clear(b)$$

and with probability 0.1, there is no change. The parameters for reward and transition probabilities may be specified with similar rules with actions and preconditions.

Probabilistic STRIPS is often more natural to use than dynamic Bayesian networks when specifying the actions in logical domains like Blocks World since an explicit factored representation is not necessary. Although probabilistic STRIPS and dynamic Bayesian networks are expressively equivalent (Littman, 1997), probabilistic STRIPS rules may represent change more compactly in some problems (see Boutilier et al., 1999, Section 4.2.3). There exists a variety of algorithms that leverage the compactness of probabilistic STRIPS representations for efficient planning (Kushmerick et al., 1995; Dearden and Boutilier, 1997).

3.7 Discussion

The previous sections of this chapter define the reinforcement learning problem where the agent must interact with a dynamic environment to maximise its expected discounted return. To summarise, the dynamics of the system may follow either a discrete-event model or a continuous-event model. SMDPs model a broad class of discrete-event dynamic systems where the events occur in continuous time. Controlled diffusion processes model continuous-event dynamic systems with stochastic differential equations. This chapter defines optimality of these models in terms of value functions, mappings from states to expected discounted return.

Although this chapter serves primarily as a review and does not present any new theoretical ideas, it synthesises and generalises existing material on discrete-event and continuous-event dynamic systems. Discrete-event dynamic systems, in

particular systems where events occur in discrete stages as opposed to continuous time, have traditionally been the subject of research in the reinforcement learning community. This chapter generalises the notation for MDPs introduced in the standard introductory text by Sutton and Barto (1998).

Few papers in the reinforcement learning literature consider continuous-event models. The papers that attempt to model continuous-event systems (e.g., Doya, 2000; Munos and Moore, 2002) typically assume that the system evolves deterministically in response to the actions of the agent.[7] Instead of using a stochastic differential equation (e.g., Equation 3.10) to describe the evolution of the system, they use standard differential equations of the form

$$ds(t) = f(s(t), a(t))dt.$$

Such systems are easier to solve since they do not involve stochastic calculus. The Hamilton-Jacobi-Bellman equation (Equation 3.11) for deterministic systems is a simpler first-order differential equation instead of a second-order differential equation. For some problems with little stochasticity, the contribution of the second order terms might be negligible, but for other problems it is important to explicitly model the stochasticity in the system to produce an effective policy.

As Section 3.3 explains, one can solve continuous-event systems by approximating the system with an SMDP. The approximation processes for deterministic and stochastic systems are similar, but the estimation of the SMDP parameters for a stochastic system is slightly more involved (Munos, 1997). One may use any of the standard SMDP solution methods, such as dynamic programming, Monte Carlo estimation, or temporal-difference learning, to extract a policy from the SMDP.

The modelling and planning system explored in this thesis uses an SMDP to approximate the dynamics of the underlying system. It does not matter whether the environment follows a discrete-event model or a continuous-event model. The world may be deterministic or stochastic without hindering the success of the proposed system. The applicability of the system to controlled diffusion processes greatly increases the utility of the system in a wide variety of domains. Controlled diffusion processes have been used in the literature to model many different problems, including financial portfolio optimisation (Korn and Kraft, 2001), production planning (Bensoussan et al., 1984), and optimal forest harvesting (Alvarez,

[7] The work by Munos and Bourgine (1998), which involves a model-based reinforcement learning algorithm for controlled diffusions, is a notable exception (see also Pareigis, 1997).

2004). The applications of controlled diffusion processes extend beyond those of discrete-event systems and deterministic continuous-event systems (see survey by Borkar, 2005).

This chapter also discusses observability. Although there are many important applications where the state of the world is only partially observable, this thesis will assume full observability. Finding solutions to partially observable problems tends to be infeasible for large problems. The system presented in this thesis may be generalised to partially observable problems through a history-dependent approach as discussed in Section 11.3.2.

The previous section describes two different ways of representing the model, dynamic Bayesian networks and probabilistic STRIPS. Both representations can compactly represent system models, and algorithms exist to exploit their structure. The model representation adopted in this thesis is different from both dynamic Bayesian networks and probabilistic STRIPS. The approach involves partitioning the state and action spaces and modelling transitions over these regions. Further details are reserved for Chapter 5.

There are ways to model and solve reinforcement learning problems that this chapter does not discuss. For example, *Gaussian processes* (Rasmussen, 2004) may be used to solve reinforcement learning problems. Rasmussen and Kuss (2004) present a model-based algorithm and Engel (2005) presents a model-free algorithm based on Gaussian processes. These approaches make strong assumptions, however, limiting their applicability to general reinforcement learning problems. The approach in this thesis does not make such strong assumptions.

The model-based methods in this chapter assume complete knowledge of the underlying system dynamics and reward function. If such knowledge is unavailable and the agent must rely upon its own experience to develop an understanding of a complex world, then the challenge becomes generalising from limited experience. The next chapter provides a broad survey of generalisation methods for reinforcement learning.

Chapter 4

Generalisation

One of the principal contributions of this thesis is a method of generalisation in large state and action spaces. This chapter provides a broad survey of generalisation methods to serve as a basis for comparison. After introducing the problem of generalisation, this chapter presents abstraction, local approximation, and global approximation techniques.

4.1 Introduction

If an agent does not have a complete understanding of the world, it must generalise from its limited experience to make intelligent decisions about how to behave. Many important domains involve large or infinite state and action spaces where it is infeasible to sufficiently explore every action from every state. The literature contains a wide variety of methods for achieving generalisation in reinforcement learning problems.

Generalisation involves exploiting structure in the representation of the state and action spaces. The assumption underlying generalisation approaches is that similarly represented states and actions behave similarly. There are different ways to represent states and actions and different ways to leverage their structure.

This chapter divides the generalisation approaches found in the literature into three broad categories. The first category involves *abstraction*, where the state and action spaces are partitioned into regions. Abstraction approaches use heuristics to decide how to partition the state and action spaces in response to experience, and they use planning algorithms such as temporal-difference learning or dynamic programming to produce suitable policies. The second category

involves *local approximation* of the value function. Local approximation uses distance metrics to generalise from learned instances in the state-action space with the assumption that instances that are close to each other have similar expected discounted return. The third category is based on *parametric approximation* of the value function.[1] As the agent acquires experience, it revises its parametric representation of the value function, typically through gradient descent. Most approaches found in the literature fall within at least one of these categories. The remainder of this chapter considers each type of approach in turn using the notation introduced in the previous chapter.

4.2 Abstraction

Abstraction is an important concept in a variety of areas within artificial intelligence, including problem solving and common sense reasoning (Giunchiglia and Walsh, 1992). This section reviews some of the existing abstraction methods for reinforcement learning in chronological order of development. Abstraction involves partitioning the state and action spaces into regions and using model-based or model-free reinforcement learning to compute an optimal policy over these regions. None of the algorithms in the literature perform abstraction in the action space,[2] although Chapman and Kaelbling (1991, Section 2) and McCallum (1995, Section 8.2.3) speculate about ways of achieving this. Other abstraction approaches not surveyed in this section include those of Reynolds (2000), Vollbrecht (2003), and Jong and Stone (2005).

The first algorithm in this section uses a human-engineered abstraction that does not change as the agent accumulates experience. The remainder of the algorithms in this section adapt their abstractions. Table 4.1 summarises some of the important properties of the adaptive abstraction algorithms. As the table reveals, these algorithms vary in their representation, planning algorithm, and splitting criterion. All of the algorithms use some form of tree structure to partition the state space. The tree structure grows incrementally when experience indicates that it should increase resolution at a particular region of the state space.

[1] Generalisation may also be achieved through parametric approximation of the optimal policy. Section 2.4.1 discusses this approach in the context of local search techniques.

[2] It is important to distinguish the action space abstractions discussed in this thesis from the *abstract actions* (also known as *options*) used in the hierarchical reinforcement learning literature (Sutton et al., 1999; Barto and Mahadevan, 2003). Abstract actions are higher-level actions specified by either closed-loop or open-loop partial policies over base-level actions.

Algorithm	Representation	Planning	Splitting Criterion
G algorithm	binary strings	TD	t-test
Parti-game	Euclidean	minimax	losing
UTree	attribute-value	DP	Kolmogorov-Smirnov
Continuous UTree	Euclidean	DP	sum-squared error
TTree	attribute-value	DP	MDL
Variable Resolution	Euclidean	DP	influence-variance
TG algorithm	relational	TD	F-test

Table 4.1: A summary of adaptive abstraction algorithms.

4.2.1 Boxes

Some problems can be solved effectively using a human-supplied discretisation of the state or action space. Michie and Chambers (1968a) discuss some early work on static state abstraction for the pole-and-cart task (see Section 2.3). They manually discretise a four-dimensional continuous state space into 255 regions, which they refer to as "boxes." Since their problem involves only two actions, there is no need to perform abstraction in the action space. They employ a Monte Carlo reinforcement learning algorithm to produce successful behaviour.

One of the main limitations of their approach is that the abstraction remains static. If the abstraction is too coarse, then the agent will not be able to find an optimal solution. If the abstraction is too fine, then the agent will take a long time to compute an optimal solution. In another paper on their Boxes algorithm, Michie and Chambers (1968b) speculate about the automatic "splitting and lumping" of regions:

> The weakness of our program in its present form is that it does not really live up to its ideal of independence of special knowledge of the physical apparatus. Some knowledge of this kind is in fact built into it when the user specifies the thresholds to be set on the state variables. It is easy to choose these in such a way as to make the control task impossible. In our plans for the next stage we aim to endow the program with the power to change the boundaries of boxes, by processes of 'splitting' and 'lumping.' (page 214)

They propose splitting regions where the best action has an expected value that is close to the expected value of other actions, and they propose merging neighbouring regions that agree which action is best. They did not implement this approach.

4.2.2 G Algorithm

Although variable resolution techniques for automatic control have been explored by others (e.g., Simons et al., 1982), the G algorithm by Chapman and Kaelbling (1991) is one of the most important early implementations, inspiring the other approaches in this section. Although there are a number of weaknesses in their approach, the idea of incrementally inducing a decision tree based on experience is useful when dealing with large state spaces.

The G algorithm assumes a fixed-length binary representation of the state space and incrementally builds a decision tree. The leaf nodes of the tree represent regions of the state space. Associated with the leaf nodes are state-action values obtained through a variation of Q-learning. The main contribution of the G algorithm is the use of Student's t-test[3] to measure the statistical significance of individual bits in the state representation. If the t-test indicates that a bit is significant in predicting either immediate reward or discounted return, the region is split on that bit. Unfortunately, when a region is split, all the information associated with that region is lost, which makes for very slow learning.

4.2.3 Parti-Game

Moore and Atkeson (1995) introduce the Parti-Game algorithm as a way to produce goal-directed behaviour in continuous states spaces. Like the G Algorithm, Parti-Game splits regions where it deems it important to do so, but the approach and assumptions are significantly different. Parti-Game assumes that the state space is represented as a vector of real values and that the agent possesses a greedy controller that can steer towards any desired state. There is no guarantee that the greedy controller will succeed, but the agent is signalled when the greedy controller becomes "stuck" against some obstacle. Parti-Game assumes that the dynamics of the world are deterministic and that the goal state is known.

The objective is to produce behaviour that takes the agent to the goal without becoming stuck, which is in contrast to the objective of traditional reinforcement learning where the agent must maximise its expected discounted return. Parti-Game uses a game-theoretic approach to decide which neighbouring region to aim towards with the greedy controller. It chooses the neighbouring region that

[3]Lehmann and Romano (2005) review the t-test and other statistical tests such as the Kolmogorov-Smirnov test used by the UTree algorithm.

minimises its estimated maximum possible cost to the goal, where the cost is the number of regions it must transition through to reach the goal. The agent maintains a database of previous experience to estimate this cost. Losing regions are regions whose minimax cost to the goal is infinite, indicating that the best policy from that region is expected to become stuck. The algorithm splits losing regions that have a non-losing neighbour and non-losing regions with losing neighbours.

Moore and Atkeson show that Parti-Game can learn competent behaviour in a variety of continuous domains with up to nine dimensions. In all of the domains they study, fewer than ten episodes are needed to learn satisfactory solutions. Unfortunately, the approach is currently limited to deterministic real-valued domains where the agent has a greedy controller. Al-Ansari and Williams (1999) and Likhachev and Koenig (2003) suggest a few improvements to the algorithm, but they do not overcome these basic limitations.

4.2.4 UTree

McCallum (1995) introduces the UTree algorithm, which extends the work by Chapman and Kaelbling (1991). The algorithm assumes an attribute-value representation of the state, where the values of the attributes are nominal. Like the G algorithm, UTree does not generalise over the action space. UTree not only makes distinctions over the current state, but it can also make distinctions based on previous observations, allowing it to handle partially observable domains.

The UTree algorithm records the experience of the agent and associates the observation, action, and reward with the appropriate leaf node in the tree. The algorithm uses these experiences to estimate the parameters of an MDP, which can be solved with dynamic programming. Periodically, the algorithm checks whether it needs to add additional distinctions to the leaves of the tree. The algorithm first computes the value $Q(o_k)$ of a particular observation o_k stored at a leaf node according to

$$Q(o_k) = R_k + e^{-\beta t_k} V(o_{k+1}), \tag{4.1}$$

where $V(o_{k+1})$ is the value of the leaf to which o_{k+1} belongs, R_k is the discounted reward between observations, t_k is the time between observations, and β is the continuous compound discount rate.[4] The algorithm uses the Kolmogorov-

[4]Equation 4.1 is a straightforward generalisation of what is contained in the thesis by McCallum (1995) to continuous-time, discrete-event systems (Section 3.2).

Smirnov statistical test to determine whether adding a distinction will result in significantly different distributions of observation values.

McCallum tests UTree on a simulated driving task. The algorithm eventually learns a better policy than the one he coded himself resulting in fewer collisions. However, UTree requires many training instances, and adding distinctions requires substantial computation.

4.2.5 Continuous UTree

Uther (2002) extends the UTree algorithm to use continuous attributes, allowing state spaces that are subsets of Euclidean space. His algorithm, however, assumes that the agent senses the underlying states of an MDP as opposed to observations emitted by a POMDP. The approach otherwise follows UTree quite closely.

To handle continuous attributes, continuous UTree considers splits between each consecutive pair of values along each dimension. Uther considers the sum-squared error splitting criterion in addition to the Kolmogorov-Smirnov test. The sum-squared error criterion involves computing the variance of the sampled state values on either side of a split. If the split significantly reduces the weighted variance, then the algorithm introduces the split into the tree.

4.2.6 TTree

Uther (2002) presents TTree as an extension to his continuous UTree algorithm. TTree learns a smaller SMDP from a generative model of a larger SMDP. A generative model of an SMDP is a randomised mapping that simulates transitions from a given state by a given action. Unlike continuous UTree and the other algorithms in this section, TTree is *not* intended to be used as an algorithm for generating intelligent behaviour from actual experience. TTree extends UTree by allowing actions with multi-step duration.

TTree samples trajectories using the generative model starting from random states within the leaf nodes. The algorithm stops sampling a trajectory when it either reaches another leaf, detects a deterministic self-transition, or exceeds some timeout. As the algorithm samples the trajectory, it tracks its accumulated discounted reward. The expected discounted return of a sampled trajectory with accumulated discounted reward R is given by $R + e^{-\beta t}V$, where t is the duration of the sampled transition and V is the value of the leaf in which the trajectory

terminates. The values of the leaves are calculated from experience using dynamic programming. The algorithm samples multiple trajectories starting from the same state to produce an estimate of the expected discounted return and optimal action from that state.

The algorithm splits a leaf node if it observes variation within a leaf of the expected discounted return for the same action. It uses minimum description length tests to decide how and when to grow the tree.

4.2.7 Variable Resolution Discretisation

Munos and Moore (2002) explore variable resolution techniques for solving reinforcement learning problems in a continuous-event dynamic system. Their techniques assume that the state space is a compact subset of Euclidean space and that the action set is finite. The world behaves deterministically, as in PartiGame, according to a differential equation. Their algorithm begins with a coarse, grid-based discretisation of the state space. The algorithm estimates an MDP from these grid points and solves the MDP using standard dynamic programming techniques.

In contrast with the other abstract algorithms in this section, the value function and policy vary linearly within each region. Munos and Moore use Kuhn triangulation (see Moore, 1992, Section 2.2.1) as an efficient way to interpolate the value function within regions. The algorithm refines its approximation by splitting cells according to a splitting criterion. Munos and Moore explore several local heuristic measures of the importance of splitting a cell including the average of corner-value differences, the variance of corner-value differences, and policy disagreement. They also explore global heuristic measures involving the influence and variance of the approximated system. The influence is a measure of non-local dependencies in the value function, and variance is an estimate of the error in the value function due to the grid approximation.

4.2.8 TG Algorithm

Driessens (2004) expands upon previous work in *relational reinforcement learning* (Džeroski et al., 2001). He presents the TG algorithm as a way of inducing logical decision trees for problems whose state spaces are described by relations between objects. A logical decision tree is a decision tree whose tests may have unbound

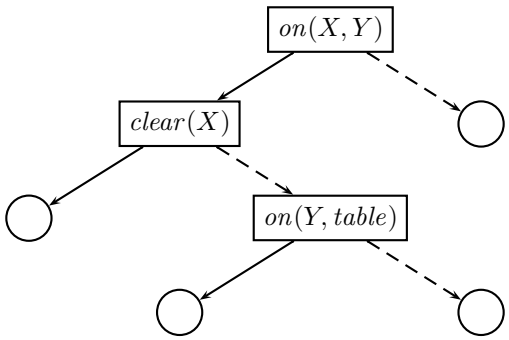


Figure 4.1: A logical decision tree. Solid lines denote true branches and dashed lines denote false branches. This example is from the Blocks World domain.

variables.[5] Figure 4.1 shows a logical decision tree with X and Y as unbound variables. Logical decision trees attempt to capture the structure inherent in the problem.

The TG algorithm is essentially the G algorithm for logical decision trees. As the agent accumulates experience, it updates the statistics at the leaf nodes for all possible new tests. The tests are logical sentences that may include variables. The sentences may introduce new variables or they may refer to variables introduced higher in the tree. The F-test indicates when there is a significant difference between the state-action values before and after a split. If the significance is above a certain threshold, the algorithm will introduce the split.

As with the G algorithm, the TG algorithm is model-free and does not memorise its experience. When the algorithm introduces a split, all of the statistics in that region of the state space must be cleared. Van Otterlo (2005) provides a recent survey of other work in relational reinforcement learning.

4.3 Local Approximation

The abstraction techniques from the previous section involve grouping states and actions together and computing optimal control over the abstraction using either model-based or model-free approaches. This section provides a brief overview of *local approximation* techniques for estimating the optimal action-value function

[5] Blockeel and De Raedt (1998) provide a detailed description of the semantics of logical decision trees and their induction.

from limited experience. These methods leverage distance metrics over the states and actions to provide generalisation with the assumption that states and actions that are close to each other behave similarly.

Local approximation techniques generally maintain a finite set of *instances*, sometimes called *prototypes*, in the state or action space. In some algorithms, these instances are actual instances experienced by the agent. Associated with these instances are estimated action values. Local approximation algorithms differ in the way they

- maintain the instances,
- use these instances to compute the action values at novel states, and
- update the action values of the instances.

There are many local approximation methods, but this section only discusses approaches that use locally weighted regression and self-organising maps.

4.3.1 Locally Weighted Regression

There are many ways to use the action values associated with stored instances to approximate the value of a novel instance. One way is to simply use the closest prototype according to some distance metric. Another way is to look at the k nearest neighbours and use the average of their values. *Locally weighted regression* calculates the value of a point based on the values and distances of the prototypes. If $\{x_1, \ldots, x_n\}$ are the prototypes, then the estimated value of a point x is

$$V(x) = \frac{\sum_{i=1}^{n} V(x_i) k(x_i, x)}{\sum_{i=1}^{n} k(x_i, x)},$$

where $k(x, x')$ is a *kernel function* or *radial basis function*. It is common to use Gaussian kernel functions of the form $e^{-(d(x,x')/h)^2}$, where h is the *bandwidth* and $d(x, x')$ is the distance from x to x' according to some metric.[6]

The Hedger algorithm (Smart and Kaelbling, 2000; Smart, 2002) applies locally weighted regression to reinforcement learning. Hedger stores state-action pairs in memory. When the agent needs to predict the value of taking an action a from a particular state s, it looks for the k closest neighbours of (s, a) within some threshold distance. If there are not k neighbours within the threshold, it returns

[6] Atkeson et al. (1997) review locally weighted regression in greater detail.

some default value. Otherwise, it computes the independent variable hull (Cook, 1979), which is a hyper-elliptic approximation of the convex hull, around the k neighbours. If (s, a) is not within this hull, it returns a default value to prevent extrapolation since regression is only valid for interpolation. If (s, a) is within the hull, the algorithm returns the locally weighted regression of the action-value function using the k neighbours. As the agent acquires experience, it adjusts the values of nearby points according to the temporal-difference error and the kernel function.

4.3.2 Self-Organising Maps

Kohonen (1982) originally proposed *self-organising maps* as an unsupervised learning approach to capture the structure of a problem.[7] A self-organising map often consists of a finite multi-dimensional grid of *units*. Associated with each unit is an initially random weight vector. A learning algorithm adjusts these weights as the agent accumulates data.

Smith (2002a,b) discusses how self-organising maps may be used for generalisation in reinforcement learning problems with large state and action spaces. He uses separate maps for the state and action spaces. The weights in the state map represent states in the state space, and the weights in the action map represent actions in the action space. Because there is only a small sampling of states and actions associated with the units in the maps, it is possible to maintain a table of state-action values. The algorithm specifies how to simultaneously update these state-action values and the weights in the maps.

When the agent encounters a state s, it searches for the closest state $\hat{s}$ in the state map. The agent then chooses the action $\hat{a}$ from the action map with the greatest value for $\hat{s}$ in the table.[8] The algorithm perturbs $\hat{a}$ slightly according to some random distribution to get a'. The agent executes action a' and transitions to state s', which is closest to $\hat{s}'$ in the state map. If t is the duration of the transition and the agent receives a discounted reward of R, the algorithm updates the weights and action values as follows:[9]

- If $R + e^{-\beta t} Q(\hat{s}', \hat{a}) > Q(\hat{s}, \hat{a})$, indicating that a' is better than $\hat{a}$, then each

[7] Kohonen (2001) provides a more recent and extensive treatment of self-organising maps.

[8] As with all reinforcement learning algorithms without a complete and accurate world model, it is necessary to periodically choose non-greedy actions.

[9] The ensuing description of the self-organising algorithm is a continuous-time generalisation of the work by Smith (2002a,b), following the notation of Section 3.2.

action $\hat{a}_j$ in the action map moves towards a' according to the rule

$$\hat{a}_j \xleftarrow{\eta_A \psi_A(\hat{a},\hat{a}_j)} a'.$$

- The value for each state-action pair $(\hat{s}_i, \hat{a}_j)$ in the action-value table moves towards the corrected value according to the rule

$$Q(\hat{s}_i, \hat{a}_j) \xleftarrow{\eta \psi_S(\hat{s},\hat{s}_i)\psi_A(\hat{a},\hat{a}_j)} R + e^{-\beta t} \max_{\hat{a}_k} Q(\hat{s}', \hat{a}_k).$$

- Each state $\hat{s}_i$ in the state map moves towards s according to the rule

$$\hat{s}_i \xleftarrow{\eta_S \psi_S(\hat{s},\hat{s}_i)} s.$$

The parameters η, η_S, and η_A are the learning rates for temporal-difference learning, state-map learning, and action-map learning respectively. The functions ψ_S and ψ_A are neighbourhood functions for the state map and action map respectively. Many applications use simple linear neighbourhood functions. For example, $\psi_S(\hat{s}_i, \hat{s}_j)$ might be given by $\max(0, 1 - (d/(n+1))$, where d is the distance between the unit with weight $\hat{s}_i$ and $\hat{s}_j$ in the grid topology. The parameter n controls the radius of the topological neighbourhood that is influenced by a weight update.

The self-organising maps proposed by Kohonen have fixed topologies, but other maps such as *Growing Neural Gas* (Fritzke, 1995) and *Grow When Required Networks* (Marsland et al., 2002) introduce new units as needed. Dynamic introduction of units is likely to be useful in reinforcement learning problems where the agent must initially generalise from very little experience and then add more units as it acquires experience to capture its expanded knowledge of the system. Millán et al. (2002), for example, use Growing Neural Gas for reinforcement learning function approximation. They show that the dynamic introduction of units is effective on real and simulated robots for tasks involving wall following and passing through doorways.

4.4 Parametric Approximation

There are a number of function-approximation methods, such as neural networks, whose function mappings are parametrically determined. Because of the need for generalisation in large state and action spaces, parametric function approximation

has been widely studied in reinforcement learning. This section briefly discusses the basic ideas underlying parametric approximation of expected discounted return.

Samuel (1959, 1967) describes one of the earliest applications of parametric value-function approximation to temporal-difference learning. He uses a vector s consisting of real-valued features to represent the state, and he estimates the value function by computing the dot product of s with the parameter vector θ. This linear approximation approach resembles that of Donaldson (1960) as described in Section 2.3. While the agent accumulates experience about the value of various board positions, a learning algorithm revises θ.

Neural networks, discussed in Section 2.3 in the context of supervised learning, provide an alternative to simple linear approximation in reinforcement learning problems. The parameter vector θ determines the weights in the neural network. If the network has at least one hidden layer, then the agent is capable of learning a non-linear approximation of the optimal value function. Bertsekas and Tsitsiklis (1996) use the term "neural dynamic programming" to describe the use of neural networks as function approximators in reinforcement problems.

In general, value-function approximation algorithms update the parameter vector θ using gradient descent. The objective is to find the point in parameter space that minimises some monotonically increasing function of the magnitude of the temporal-difference error. The backpropagation algorithm (Rumelhart et al., 1986) is one way to perform this gradient descent to find a locally optimal value for θ.

Lin (1992) approximates the value function with a multi-layer neural network. The input to the neural network is a factored representation of the state. The network has multiple outputs, one for each action. Lin also suggests that a single network can be replaced by multiple networks, one for each action, each with a single output unit. Generalisation is not done over actions. Lin uses backward experience replay to enhance performance. Rummery and Niranjan (1994) use neural networks with other reinforcement learning algorithms including Q-learning (Watkins, 1989) and Q(λ) (Peng and Williams, 1996). Anderson (1986) combines neural networks with the Adaptive Heuristic Critic algorithm (Sutton, 1984).

Various researchers (e.g., Gullapalli, 1990; Baird and Klopf, 1993; Santamaría et al., 1997) explore the use of neural networks for generalisation in the action

space as well as the state space. For example, Santamaría et al. (1997, Section 3.1) use factored representations of both the state and action space as input to a single neural network. To estimate the best action from a particular state, they randomly sample actions and choose the one that produces the greatest output from the neural network. Doya (2000) also explores generalisation in both state and action spaces, but instead of considering function approximation in discrete-time systems, he focuses on approximation for continuous-event problems, such as those described by differential equations.

There have been a number of successful applications of parametric function approximation. Tesauro (1992), for example, describes a temporal difference algorithm that uses neural network function approximation to learn to play backgammon at a human-competitive level. However, value-function approximation using gradient descent on the temporal-difference error can lead to divergence in certain situations (Thrun and Schwartz, 1993; Boyan and Moore, 1995). Baird (1999) suggests an approach that guarantees convergence. Instead of performing gradient descent only on the temporal-difference error, he proposes minimising a mixture of the temporal-difference error with an estimate of the sum-squared Bellman residual.

4.5 Discussion

The previous three sections survey generalisation through abstraction, local approximation, and parametric approximation. This section discusses these three approaches in relation to each other and explains how they pertain to the approach taken in this thesis.

The abstraction methods partition the state space into regions. For difficult problems, it is important that the partitioning be dynamic. As the agent accumulates experience, it should incrementally refine its discretisation of the state space. As Section 4.2 shows, there are many different approaches for deciding how and when to refine a discretisation. With the exception of Parti-Game (Moore and Atkeson, 1995), the abstraction methods in the literature consider every possible way of splitting a region and choose the split the produces the most statistically significant difference in the estimated distribution of value in the resulting regions.

None of the abstraction methods in Section 4.2 perform abstraction in the ac-

tion space, although some of the authors recognise that generalising over actions is important. In problems with large or continuous action spaces, generalisation is critical to success. The approach taken in this thesis appears to be the first to implement abstraction in both the state and action spaces. When performing abstraction in both spaces, the algorithm must decide whether to introduce distinctions in the state or action space, as Chapter 6 discusses. The abstraction methods in the literature also only increase resolution over time. They do not have provision to decrease resolution when further experience indicates that doing so is beneficial. Chapter 7 explains how simplification of the model might be accomplished.

The previous two sections present local and parametric approximation approaches to generalisation. Local approximation approaches perform generalisation based on locality as measured by some distance metric. Local approximation uses stored instances in the state-action space to locally represent the value function. In contrast, parametric approximation uses a parameter that it revises, typically through gradient descent, to globally represent the value function. The literature contains examples of local and parametric approximation techniques performing generalisation in both the state and action spaces.

Local and parametric approximation approaches do not build models of the system dynamics. They typically rely upon some form of temporal-difference learning to estimate the value function and produce a plan.[10] Abstraction techniques, on the other hand, may utilise either temporal-difference learning or dynamic programming because they allow the construction of models from experience.[11] Local and parametric approximation are at a disadvantage to abstraction because temporal-difference learning, even with eligibility traces, requires much more experience than model-based approaches to produce suitable policies. Because abstraction better utilises limited experience, the remainder of this thesis focuses primarily on abstraction as a method of generalisation.

[10] One may use a dynamic programming approach similar to value iteration with local and parametric approximation, but the computational complexity of each update is generally proportional to the number of samples (see Ormoneit and Sen, 2002).

[11] The G algorithm (Chapman and Kaelbling, 1991) and related algorithms (e.g., Driessens, 2004) do not produce models, but adapting them to use a model-based planning system is not difficult.

Chapter 5

Modelling

This chapter motivates and explains modelling in AMPS. Modelling involves updating a representation of the system dynamics as the agent accumulates experience. AMPS generalises from experience by partitioning the state and action spaces into regions to form a map and estimating the parameters of an SMDP over this map. The first section introduces the approach, the second section discusses splitting and merging regions, and the third section discusses estimation. The fourth section discusses the assumptions AMPS makes when estimating abstract models and issues with termination. The fifth section discusses exploration strategies and the incorporation of guidance. The final section summarises the contributions made in this chapter.

5.1 Introduction

For an agent to be considered intelligent, it must have some understanding of the world. This understanding is generally in the form of a model. The model captures an approximation of the relevant system dynamics and may be used to predict outcomes of behaviour. Certainly, humans and some other animals are able to construct at least some form of predictive model, and it is difficult to imagine an artificial agent functioning successfully in the real world without one.

As established earlier in this thesis, there have been many successful applications of reinforcement learning without the utilisation of an explicit model of the system dynamics. However, such model-free approaches generally require much more time to produce satisfactory behaviour than model-based approaches. The superiority of model-based approaches has been confirmed experimentally (e.g.,

Moore and Atkeson, 1993; Peng and Williams, 1993; Atkeson and Santamaría, 1997) and is understandable because model-free approaches can only update the values of states along a single trajectory. If the goals or reward structure changes, model-free approaches are at an even greater disadvantage. Without a model, learning must start from scratch, but if the agent has a model, it may replan over the model.

One of the challenges of applying model-based learning is deciding how to model the world dynamics. If the state and action spaces are large, then it is necessary to incorporate generalisation. The most natural form of generalisation that is suitable for model-based approaches is abstraction (Chapter 4). Abstraction involves partitioning the state and action spaces into regions to form a *map*. SMDPs are suitable models of the dynamics over these regions because they handle uncertainty in state transition, duration, and reward.[1] Such uncertainty arises as a result of grouping together states and actions and the underlying stochasticity in the world.

Modelling involves using experience to update the map and the estimates of the SMDP parameters over this map. The next section discusses map revision and the following section discusses estimation. Because the agent must revise its map and update its estimates continuously as it interacts with the world, efficiency is an important concern and a major consideration in the design of AMPS.

5.2 Map Revision

This section addresses how to use the experience accumulated by the agent to revise its map efficiently and effectively. It is important to realise that there is no single "correct" way of partitioning the state and action spaces since any partitioning leads to a sensible SMDP. Therefore, it is necessary to rely upon heuristics to guide the adaptation of the map. AMPS adapts the partitions of the state and action spaces by splitting and merging regions according to a set of heuristics. AMPS also uses heuristics to prioritise revisions to the map. When time is available, the agent revises the highest priority regions. Chapters 6 and 7 describe these heuristics in depth.

[1] Benson (1996) explores an alternative to SMDPs for problems involving goals instead of general reward functions. He uses inductive logic programming (Muggleton and De Raedt, 1994) to learn action models in the form of teleo-operators (Benson and Nilsson, 1995).

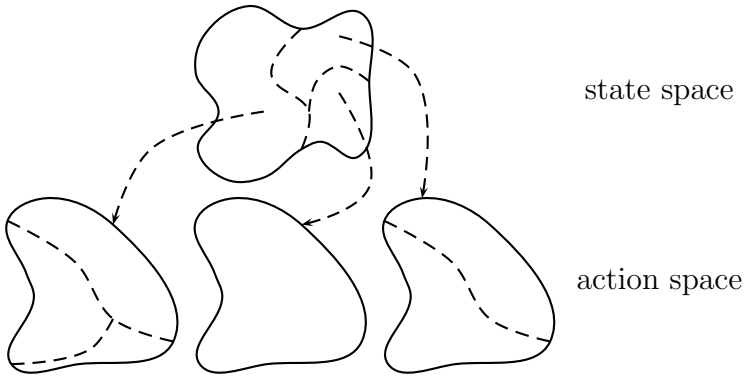


Figure 5.1: The relationship between state partitions and action partitions. Each region in the state space has its own partition of the action space. The region on the left side of the state space defines a partition of the action space with three regions, the region on the bottom right of the state space defines a partition with a single region, and the region on the top right of the state space defines a partition with two regions.

The map in AMPS defines a separate partition for each state region as Figure 5.1 illustrates. As Section 4.3 describes, other researchers (e.g., Smith, 2002a) maintain only a single discretisation of the action space. It is useful for each state region to define its own partition of the action space because it allows greater generalisation. In some regions, it is necessary to very finely distinguish between actions, but in other regions such high resolution distinctions are not useful.

This thesis uses the following notation when discussing maps:

- $\mathcal{S}$ is the partition of the state space,
- $S(s)$ is the state region to which state s belongs,
- $\mathcal{A}(S)$ is the action partition associated with the state region S, and
- $A(\mathcal{A}, a)$ is the action region in the partition $\mathcal{A}$ to which action a belongs.

For convenience, $A(s, a)$ is shorthand for $A(\mathcal{A}(S(s)), a)$, which gives the action region associated with action a from state s. Whatever data structure is chosen to implement the map, it is important that it be able to efficiently map states to their state regions and state-action pairs to their action regions. In addition, it is important that the data structure supports an efficient implementation of the following operations for revising the map:

- SPLIT$(S, S_1, \ldots, S_n)$, splits state region S into some number of new regions such that each set of states $S_1, \ldots, S_n \subset S$ becomes part of separate state regions,

- SPLIT$(A, A_1, \ldots, A_n)$, splits the action region A into some number of new regions such that each set of actions $A_1, \ldots, A_n$ becomes part of separate action regions,

- MERGE$(S_1, \ldots, S_n)$, merges the state regions $S_1, \ldots S_n$, and

- MERGE$(A_1, \ldots, A_n)$, merges the action regions $A_1, \ldots, A_n$.

The subscript n in the description above may vary. For the SPLIT operations, the data structure is not obligated to split the sets of states perfectly. In fact, it might be desirable to not split the sets of states perfectly to prevent *overfitting*, which is a common problem in supervised learning. The implementation of the SPLIT operations actually performs a form of supervised learning. Splitting the sample states or actions involves learning a multi-way classifier that classifies the samples into different categories.

It is important that the data structures supporting the map be efficient and support arbitrary representations of states and actions. The current version of AMPS supports two kinds of implementations of the map, one based on decision graphs and the other based on nearest neighbour generalisation. These two kinds of data structures are particularly well-suited for AMPS because they can perform splits and merges efficiently and may be adapted to different kinds of state and action representations. In some domains, it is useful to combine different data structures to partition the different spaces. For example, nearest neighbour might partition the state space, but a decision graph might partition the action space.

The splitting operations require processing the underlying state and action representations. Because representation is key to successful generalisation, it is important to keep all representation-dependent implementation in a small, self-contained module that may be tailored to leverage the representation. The decision graph representation must have access to a module that produces tests, either binary or multi-valued, from examples. The nearest neighbour implementation must have access to a module that computes distances between two states or two state-action pairs. Details of how decision graphs and nearest neighbour

support separating and merging states and actions are reserved for Chapters 6 and 7.

5.3 Estimation

This section explains how to estimate the parameters of an SMDP from experience. The strategies presented in this section are used by AMPS to model the transition and reward dynamics over the partitioned state and action spaces. AMPS assumes that the transitions and rewards are received according to an SMDP over the regions, which might not actually be the case. In reality, the dynamics over the partitioned spaces may be non-Markovian (see Section 3.5). This section presents strategies for estimating $P(s' \mid s, a)$, $R(s, a)$, and $\gamma(s, a, s')$ from experience with the assumption that the model actually follows an SMDP.

An important concept in this section is that of *maximum-likelihood estimation* (Fisher, 1922). Maximum-likelihood estimation computes the most likely value or parameter setting given the data.[2] The maximum-likelihood estimate for $P(s' \mid s, a)$ is easy to compute. Let $n(s, a)$ be the number of times action a was taken in state s, and let $n(s, a, s')$ be the number of times action a was taken in state s resulting in a transition to state s'. The maximum-likelihood estimate for $P(s' \mid s, a)$ is

$$\hat{P}(s' \mid s, a) = n(s, a, s')/n(s, a).$$

It can be shown by the strong law of large numbers that $\hat{P}$ will converge almost surely to the true P, assuming that the agent follows an appropriate exploration policy such that over time $n(s, a, s')$ approaches infinity.

Estimating $R(s, a)$ and $\gamma(s, a, s')$ is slightly more involved because they depend upon $r(s, a, s')$, $\rho(s, a, s')$, and $P_t(t \mid s, a, s')$. There are two different ways to estimate R and γ. The first method involves computing $\hat{P}_t$ by estimating the parameters of an assumed distribution model. The second method does not require prior knowledge of the distribution model for P_t; instead $\hat{R}$ and $\hat{\gamma}$ are estimated directly. Both methods have their advantages and disadvantages.

[2] Duda et al. (2000, Chapter 3) provide an introduction to maximum-likelihood estimation and other estimation techniques.

5.3.1 Parametric Model Estimation

If P_t follows a known parameterised distribution, then maximum likelihood can estimate its parameters. With an estimate of the distribution, it is possible to integrate Equations 3.2 and 3.3 to estimate γ and λ. It is possible to estimate R by substituting the estimates of γ and λ along with the maximum-likelihood estimates of r and ρ into Equation 3.4. The estimates for R and γ will strongly converge to their true values.

For an example of how one might use parametric model estimation, suppose P_t follows an exponential distribution.[3] The cumulative distribution function of the exponential distribution is $1 - e^{-\theta t}$ and the probability density function is $\theta e^{-\theta t}$. If the agent has made n transitions from s to s' by taking action a and the durations of these transitions are given by $t_1, \ldots, t_n$, the likelihood of parameter θ is proportional to the product of the density at each sample point:

$$L(\theta) \propto \prod_{k=1}^{n} \theta e^{-\theta t_k}.$$

Computing the parameter that maximises this likelihood involves setting the derivative of the log-likelihood, i.e. $\partial \ln L(\hat{\theta})/\partial\hat{\theta}$, to zero and solving for $\hat{\theta}$. Incidentally, $\hat{\theta}$ is simply the inverse of the sample mean. So, if $\sigma_t(s, a, s')$ is equal to the total time spent transitioning from s to s' by action a, then $\hat{\theta}(s, a, s') = n(s, a, s')/\sigma_t(s, a, s')$. Therefore,

$$\begin{aligned}
\hat{\gamma}(s, a, s') &= \int_0^\infty e^{-\beta t} dP_t(t \mid s, a, s') \\
&= \hat{\theta}(s, a, s') \int_0^\infty e^{-t(\beta + \hat{\theta}(s,a,s'))} dt \\
&= \frac{\hat{\theta}(s, a, s')}{\beta + \hat{\theta}(s, a, s')} \\
&= \frac{n(s, a, s')}{\sigma_t(s, a, s')\beta + n(s, a, s')}.
\end{aligned}$$

Computing $\hat{R}(s, a, s')$ follows a similar process. Let $\sigma_r(s, a, s')$ be the total lump-sum reward received when transitioning from s to s' by action a, and let $\sigma_\rho(s, a, s')$ be the sum of all the rates received when transitioning from s to s' by action a. The maximum-likelihood estimates of $r(s, a, s')$ and $\rho(s, a, s')$ in the

[3] When P_t follows an exponential distribution, the SMDP is called a continuous-time Markov decision process (CTMDP) as discussed by Howard (1971, Chapter 10) and Puterman (1994, pages 530–531).

equation for R (Equation 3.4) are $\sigma_r(s,a,s')/n(s,a,s')$ and $\sigma_\rho(s,a,s')/n(s,a,s')$, respectively. By the strong law of large numbers, these estimates strongly converge to their true values. Combining the estimate for γ with Equation 3.6 produces the estimate

$$\begin{aligned}\hat{\lambda}(s,a,s') &= (1-\hat{\gamma}(s,a,s'))/\beta \\ &= \left(1-\frac{n(s,a,s')}{\sigma_t(s,a,s')\beta+n(s,a,s')}\right)\left(\frac{1}{\beta}\right) \\ &= \frac{\sigma_t(s,a,s')}{\sigma_t(s,a,s')\beta+n(s,a,s')}.\end{aligned}$$

The following formula estimates R, omitting "(s,a,s')":

$$\begin{aligned}\hat{R}(s,a) &= \sum_{s'\in\mathbb{S}}\hat{P}(s'\mid s,a)(\hat{\gamma}\hat{r}+\hat{\lambda}\hat{\rho}) \\ &= \sum_{s'\in\mathbb{S}}\left(\frac{n}{n(s,a)}\right)\left[\left(\frac{n}{\sigma_t\beta+n}\right)\left(\frac{\sigma_r}{n}\right)+\left(\frac{\sigma_t}{\sigma_t\beta+n}\right)\left(\frac{\sigma_\rho}{n}\right)\right] \\ &= \frac{1}{n(s,a)}\sum_{s'\in\mathbb{S}}\frac{n\sigma_r+\sigma_t\sigma_\rho}{\sigma_t\beta+n}.\end{aligned}$$

The estimate $\hat{R}$ strongly converges to its true value.

Instead of using maximum-likelihood estimation, which is a frequentist approach, one may use a Bayesian approach.[4] The Bayesian approach provides a probability distribution over the unknown parameters instead of point estimates. Under the assumption that transition durations follow an exponential distribution,

$$\begin{aligned}\hat{\gamma} &= \int_0^\infty\int_0^\infty e^{-\beta t}\theta e^{-\theta t}dtp(\theta)d\theta \\ &= \int_0^\infty\frac{\theta}{\theta+\beta}p(\theta)d\theta\end{aligned}$$

with "(s,a,s')" omitted. The density $p(\theta)$ is estimated from experience according to Bayes' rule assuming some prior distribution over θ. If the durations follow an exponential distribution with parameter θ, then it is convenient to specify the prior over θ in terms of the conjugate prior of the exponential distribution, namely the gamma distribution. The density of the gamma distribution parameterised by θ_1 and θ_2 is given by

$$p(\theta;\theta_1,\theta_2)=\frac{\theta_2^{\theta_1}}{\Gamma(\theta_1)}\theta^{\theta_1-1}e^{-\theta\theta_2},$$

[4] Bolstad (2004) and Jaynes (2003) provide introductions to Bayesian statistics and compare the frequentist and Bayesian views of probability.

where $\Gamma(\cdot)$ is the gamma function. Given n and σ_t as defined earlier and a prior gamma distribution with parameters θ_1 and θ_2, the posterior density is given by a gamma distribution with parameters $\theta_1 + n$ and $\theta_2 + \sigma_t$. Hence,

$$\begin{aligned}\hat{\gamma} &= \int_0^\infty \left(\frac{\theta}{\theta+\beta}\right) \frac{(\theta_2+\sigma_t)^{(\theta_1+n)}}{\Gamma(\theta_1+n)} \theta^{(\theta_1-1)} e^{-\theta(\theta_2+\sigma_t)} d\theta \\ &= \frac{(\theta_2+\sigma_t)^{(\theta_1+n)}}{\Gamma(\theta_1+n)} \int_0^\infty \frac{\theta^{\theta_1} e^{-\theta(\theta_2+\sigma_t)}}{\theta+\beta} d\theta.\end{aligned}$$

When commencing with a diffuse prior over θ, $\hat{\gamma}$ is simply

$$\frac{\sigma_t^{n+1}}{n!} \int_0^\infty \frac{\theta^{n+1} e^{-\theta\sigma_t}}{\theta+\beta} d\theta.$$

Unfortunately, this integral must be computed numerically. Because of the added complexity of the Bayesian calculation of $\hat{\gamma}$, this thesis focuses on maximum-likelihood estimation. As n goes to infinity, the maximum-likelihood estimate converges to the Bayesian estimate.

5.3.2 Non-Parametric Model Estimation

This section considers the problem where P_t follows some arbitrary unknown distribution. Instead of estimating this distribution, it is easier to estimate γ and R directly.

Calculating γ and R involves integrating over probability distribution functions. Monte Carlo integration (see Gentle, 2002, Section 2.2) is one way to evaluate the integrals that define γ and λ (Equations 3.2 and 3.3). Monte Carlo integration approximates the evaluation of the definite integral involving the function $g(x)$ and cumulative distribution function $F(x)$ as follows:

$$\int_{-\infty}^{\infty} g(x) dF(x) \approx \frac{1}{n} \sum_{k=1}^{n} g(x_k),$$

where the samples $x_1, \ldots, x_n$ are selected from $F(x)$. The estimate converges as the number of samples increases. The agent may incrementally update its estimate of γ after observing the completion of the kth transition and updating $n(s_k, a_k, s_{k+1})$:

$$\hat{\gamma}(s_k, a_k, s_{k+1}) \leftarrow \hat{\gamma}(s_k, a_k, s_{k+1}) + \left(e^{-\beta t_k} - \hat{\gamma}(s_k, a_k, s_{k+1})\right) / n(s_k, a_k, s_{k+1}).$$

The agent may use $\hat{\gamma}$ to estimate R. After some algebraic simplification and omitting "(s, a, s')"

$$\hat{R}(s,a) = \frac{1}{n(s,a)} \sum_{s' \in \mathbb{S}} \left[\hat{\gamma}(\sigma_r - \sigma_\rho/\beta) + \sigma_\rho/\beta\right].$$

An alternative non-parametric estimation method involves estimating $\hat{R}(s, a)$ by averaging the samples of the discounted reward (Equation 3.1). Because the discounted reward involves products of random variables, this estimation technique is less efficient (Goodman, 1960).

5.3.3 Interrupted Trajectories Estimation

It may be the case that the agent decides to change the action that it was taking before the transition to another state completes. This situation might occur if planning occurs during the execution of an action. It is possible that a value update causes the optimal policy to change at the current state, in which case the agent must decide whether to continue its possibly suboptimal action or change its action to be consistent with the updated policy. Another situation where the agent might wish to change actions is when the execution of a particular action is taking an unusually long time, perhaps beyond some set threshold, to complete its transition.

When a trajectory is interrupted because of a change in action, the agent can and should use the information gained from the experience up to the point of interruption. Although the agent does not receive reward for an interrupted transition, the agent knows the duration of time it spent in the transition before changing its action. Unfortunately, it is not clear how to update $P_t(t \mid s, a, s')$ since the probability depends on the state to which the agent would have transitioned if the trajectory was not interrupted. The remainder of this section assumes that $P_t(t \mid s, a) = P_t(t \mid s, a, s')$ for all s' and that P_t follows a parameterised distribution.

Care must be taken when updating the estimate of P_t because the actual duration of the completed transition is not known, only that it is greater than the amount of time spent in the transition before interruption. As in Section 5.3.1, assume that P_t follows the exponential distribution with parameter θ. Let $\{t_1, \ldots, t_n, \underline{t}_1, \ldots, \underline{t}_m\}$ be the set of measurements of P_t, where the t_k's are completed durations and the $\underline{t}_k$'s are incomplete durations. Also assume that $n > 0$. Finding the most likely value of θ under these conditions frequently appears in reliability analysis. The $\underline{t}_k$'s can be thought of as "time censored" or "Type I censored" data censored to the right (Meeker and Escobar, 1998, pages 34–39). The likelihood of a parameter θ is proportional to the product of

the density at each t_k and the tail probability to the right of each $\underline{t}_k$:

$$L(\theta) \propto \left[\prod_{k=1}^{n} \theta e^{-\theta t_k}\right] \left[\prod_{k=1}^{m} e^{-\theta \underline{t}_k}\right].$$

Setting the derivative of the log-likelihood to zero and solving for θ gives the maximum-likelihood estimate of θ,

$$\hat{\theta} = n / \left[\sum_{k=1}^{n} t_k + \sum_{k=1}^{m} \underline{t}_k\right],$$

or, in other words, the number of times the transition has completed successfully divided by the total time spent in both complete and incomplete transitions.

If $\sigma_t(s, a)$ denotes the total time spent in complete and incomplete transitions from s by action a and $n(s, a)$ denotes the number of complete transitions from s by action a, then

$$\hat{\gamma}(s, a) = \frac{n(s, a)}{\sigma_t(s, a)\beta + n(s, a)}$$

and

$$\hat{R}(s, a) = \frac{\sigma_r(s, a) + \sigma_t(s, a)\sigma_\rho(s, a)/n(s, a)}{\sigma_t(s, a)\beta + n(s, a)}.$$

The simplicity of this estimate is a property of the exponential distribution; other distributions may require more complex calculations.

5.4 Abstract SMDPs

The previous section explained parametric and non-parametric estimation for SMDPs. This section discusses modelling assumptions and issues with termination in abstract SMDPs.

5.4.1 Assumptions

A transition from S to S' is a trajectory through states in S to some state in S'. Hence, abstract SMDPs do not contain self-loops. In other words, $P(S \mid S, A) = 0$ for all state regions S and action regions A.

Since the agent is able to sample the world only at some finite frequency, certain assumptions are necessary about what occurs between samples when estimating the model parameters. AMPS assumes that when transitioning from S to S', the first state sample in S' is the first state in S' the agent encountered.

Hence, when estimating the duration required to transition from S to S', AMPS uses the duration of time from the first sample in S until the first state sample in S'.

5.4.2 Termination

An episode may terminate naturally or unnaturally. *Natural termination* is due to the interaction of the agent in the world, perhaps with the achievement of some goal or the encountering of some fatal state. *Unnatural termination* occurs because of some external cause, such as a specified timeout or manual system reset. The agent should not explicitly model unnatural termination, but it should handle natural termination.

There are different ways to handle natural termination. One way is to add an *absorbing state*, say s_0, where $P(s_0 \mid s_0, a) = 1$ for all actions a. In other words, once the agent reaches the state s_0, it cannot exit. The agent receives zero reward indefinitely once reaching an absorbing state.

The state s_0 is *synthetic*; it does not correspond to an actual sensor reading. A transition from state region S to state s_0 consists of the trajectory in S until termination in S. It might be the case that the episode terminates immediately upon arrival to some region S. In this case, a duration of zero contributes to the estimate of time required to transition from S to s_0. Only transitions to s_0 may be deterministically instantaneous, meaning $P_t(0 \mid S, A, s_0)$ may be 1 for some region S.

5.5 Exploration

In order for the agent to construct a useful model, it must perform sufficient exploration. This section discusses various exploration strategies. Because exploration by itself is unlikely to result in satisfactory behaviour during the early stages of learning, this section also discusses methods for incorporating guidance to improve performance.

5.5.1 Strategies

The choice of exploration strategy greatly impacts performance. Too little exploration prevents the agent from learning an accurate model, thereby hindering the

development of a satisfactory policy. Too much exploration, on the other hand, limits the exploitation of a good policy. As Section 2.6.2 discusses, determining an optimal Bayesian exploration strategy in a large environment is not feasible and requires the specification of a prior distribution over models (cf. Dearden et al., 1999). Therefore, it is common in adaptive problems to rely upon a heuristic exploration strategy. Exploration strategies that balance exploration of the state space with exploitation of good policies have been considered extensively in the literature for MDPs (e.g., Thrun, 1992; Meuleau and Bourgine, 1999), and many of the strategies are also suitable for exploration in SMDPs. Exploration strategies fall into two categories, undirected strategies and directed strategies.

Undirected Exploration

An *undirected exploration strategy* relies on randomness for exploration. One of the most popular exploration strategies is *ϵ-greedy exploration*. With some typically small probability ϵ this strategy selects a random action and the remainder of the time it selects a greedy action.

Another popular exploration strategy is *softmax* action selection where actions are selected according to some probability distribution related to their value $Q(s, a)$. It is common to use the *Gibbs-Boltzmann distribution* with softmax. Action a is selected in state s with probability

$$P(a \mid s) \propto e^{Q(s,a)/\tau},$$

where τ is the "temperature" parameter that controls the level of exploration. Increasing τ increases exploration. When τ approaches zero, the resulting strategy approaches the greedy strategy.

Directed Exploration

A *directed exploration strategy* utilises information about past exploration to guide its search. Directed strategies tend to perform better than undirected strategies because they focus exploration in areas where there is greater uncertainty in the model. Some strategies use information about how frequently or recently actions have been taken in the past. Some approaches estimate upper bounds on $Q(s, a)$ and select actions optimistically, as done by Wiering and Schmidhuber (1998) in their model-based interval estimation strategy based on work by Kaelbling (1993,

Chapter 4). In the Dyna system, Sutton (1990) provides exploration bonuses to actions whose qualities are uncertain.

The experiments in Chapter 8 use an exploration heuristic similar to the one used by Moore and Atkeson (1993). With $n(s,a)$ representing the number of times action a was taken in state s and given some parameter n_{bored}, $Q^{\max}(s,a)$ is defined as follows:

$$Q^{\max}(s,a) = \begin{cases} V(s) & \text{if } n(s,a) < n_{\text{bored}} \\ Q(s,a) & \text{otherwise} \end{cases} . \tag{5.1}$$

One may use $Q^{\max}$ in place of Q in ϵ-greedy and softmax selection. So long as the agent selects action a fewer than n_{bored} times, $Q^{\max}$ makes action a appear as good as any other action. This exploration strategy encourages random exploration until all actions have been selected n_{bored} times.

This thesis does not aim to advance the state of the art in balancing exploration with exploitation. Other researchers (e.g., Brafman and Tennenholtz, 2002; Kearns and Singh, 2002) have investigated more sophisticated algorithms that provide performance guarantees under certain assumptions. These algorithms are only applicable to MDPs with finite state and action spaces. Chapter 9 describes a novel action-selection algorithm appropriate for abstract SMDPs whose partition of the state and action spaces evolve over time.

5.5.2 Guidance

If the agent begins with a *tabula rasa* understanding of the problem and does not receive guidance from an external source, the agent must rely on random exploration. In difficult problems, random exploration is not likely to provide the experience necessary for constructing a useful model, especially in problems where the reward structure is uniform except at a few goals. In such environments, the agent flails about randomly until reaching a goal. Depending on the problem, reaching a goal by trial and error can take a very long time.

Whitehead (1991) discusses the impracticality of initial random search and motivates the use of guidance in reinforcement learning agents. He says,

> ...in nature, intelligent agents do not exist in isolation, but are embedded in a benevolent society that is used to guide and structure learning. Humans learn by watching others, by being told, and by receiving criticism and encouragement. *Learning is more often a transfer than a discovery.* Similarly, intelligent robots cannot be expected

> to learn complex real-world tasks in isolation by trial and error alone. Instead, they must be embedded in cooperative environments, and algorithms must be developed to facilitate the transfer of knowledge among them. Within this context, trial-and-error learning continues to play a crucial role: for pure discovery purposes and for refining and elaborating knowledge acquired from others. (page 607)

Observing how another agent approaches a problem can be tremendously useful in revealing the regions of the state space that are likely to be useful. In Chapter 10, AMPS and other agents are initially guided by imperfect teachers. It should be noted that guidance in reinforcement learning is very different from the supervised learning approach for behavioural cloning (Section 2.3). Because reinforcement learning does not attempt to mimic the behaviour of the teacher, it is more robust than behavioural cloning to imperfect teachers. Although a reinforcement learning agent is more likely to learn useful behaviour from a competent teacher, an incompetent teacher does no long-term harm.

Many other researchers have studied the incorporation of guidance during the early stages of learning. For example, Lin (1991) describes experiments with a reinforcement learning system for controlling a robot whose task involves docking with a battery charger. In his experiments, the reinforcement learning system by itself was unable to solve the task, but with a single demonstration by a teacher, the system quickly learned successful behaviour. In other work described by Smart and Kaelbling (2000), only a single training episode is required to train their reinforcement learning system to solve the mountain-car task.

Other researchers have explored the interleaving of random exploration with guided exploration. Early work by Chambers and Michie (1969) explored mechanisms for interleaving human assistance with their reinforcement learning algorithm. Driessens and Džeroski (2004) explore strategies for relational reinforcement learning where the agent is permitted to ask for guidance from a teacher starting in states from which it has previously failed to find a path to the goal.

Clouse (1996) describes a system where the agent queries a teacher when it is uncertain about how to proceed in a task. The agent uses the difference between the highest and lowest action value from the current state to measure its uncertainty. If the uncertainty is above a certain threshold, the agent asks for assistance. AMPS uses an approach similar to that of Clouse when a teacher or "oracle" is present. If there does not exist a path in the model from the current region to a goal, then AMPS will query the oracle.

Having a teacher guide the agent through a few training episodes is perhaps the easiest way to supply a reinforcement learning algorithm with enough information from which it can successfully bootstrap. The remainder of this section discusses alternative ways to incorporate prior knowledge so that the agent does not have to rely solely upon random exploration.

One way to make reinforcement learning easier is by adjusting the reward structure. For example, to make goal-achievement tasks easier to solve, an expert might introduce small artificial rewards in the environment to encourage the agent to pursue a path to the goal. Adjusting the reward structure to make reinforcement learning easier is called *shaping* (Mataric, 1994; Randløv and Alstrøm, 1998; Laud and DeJong, 2003).

When using shaping, it is important that the restructured reward function leads to an optimal solution that is consistent with the original reward function. Ng et al. (1999) show how to transform reward by means of potential functions without changing the optimal policy. Wiewiora et al. (2003) extend the work by Ng et al. on potential-based shaping.

Another way of incorporating prior knowledge into reinforcement learning is in the initialisation of the value function and model. Most of the work on using prior knowledge to initialise the value function involves model-free algorithms. Carroll et al. (2001) explore several different ways of transferring knowledge through estimates of the action-value function. Wiewiora (2003) shows that potential-based shaping is equivalent to action-value initialisation in Q-learning (Watkins, 1989). One problem with providing an initialisation of the value function or model is that the bias induced by poor initialisations can be difficult to unlearn.

Andre and Russell (2002) attempt to reduce the time a reinforcement learning agent spends exploring by reducing the search space. In their approach, an expert encodes a hierarchical partial program specifying how the agent should behave. The agent is given choices at certain points in the program. Other hierarchical reinforcement learning algorithms have been explored by Parr (1998), Sutton et al. (1999), and Dietterich (2000).

5.6 Discussion

This section summarises and discusses the contributions of this chapter to incremental model construction from experience. Incremental model construction

involves map revision and estimation, and AMPS takes a novel approach in both areas.

AMPS constructs its model of the system dynamics by incrementally revising its abstraction of the problem. Its abstraction involves using a map, implemented with a decision graph or nearest neighbour classifier, to partition the state and action spaces. It is not practical to consider every possible way of partitioning experienced states,[5] so AMPS only locally revises the partition by splitting or merging existing regions. As Chapter 4 explains, AMPS is not unique in pursuing an incremental abstraction approach, but AMPS is unique in how it combines both splitting and merging in both the state and action spaces. The next two chapters discuss splitting and merging in greater detail. These chapters explore possible implementations of the split and merge operations that this section defines abstractly.

The abstract definition of the split and merge operations is novel and is one of the most interesting contributions of this thesis. When AMPS decides that it needs to split a region, it presents the mapping module with sets of examples within that region that should be separated. Such an approach connects reinforcement learning to a form of supervised learning since map revision involves classifying labelled examples. In contrast with most supervised learning scenarios, however, a teacher does not supply the labelled examples. Instead, the labelled examples come from an unsupervised process responsible for clustering states and actions (Chapter 6).

The way in which AMPS combines ideas from supervised and unsupervised learning to solve reinforcement learning problems is novel. The approach taken in AMPS is useful because it allows the general system to be applied to a wide variety of domains. Only the map requires adjustment to exploit the structure embedded in the underlying representation of the states and actions. The next chapter explains how to leverage representations using nearest neighbour and decision graph approaches.

The estimation techniques presented earlier in this chapter are not new. Maximum-likelihood and Monte Carlo estimation have been studied extensively in a variety of settings. This chapter applies these techniques to estimate the parameters of an SMDP, presenting both parametric and non-parametric estimation

[5] The number of ways of partitioning a set of size n is denoted by the Bell number B_n (Bell, 1934; Rota, 1964), which grows according to the recurrence $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$.

strategies. This chapter also explains how to utilise information from interrupted transitions. Although the general theory for handling "time censored" or "Type I censored" data censored to the right is well-known within some subfields such as reliability analysis, AMPS appears to be the first to exploit censored data in a reinforcement learning context. When trajectories are frequently interrupted, it is absolutely critical to leverage censored data as the experiments in Section 8.5.3 demonstrate.

When trajectories are interrupted, other reinforcement learning algorithms either treat censored measurements as uncensored measurements or ignore censored measurements entirely. The TTree algorithm (Uther, 2002; Uther and Veloso, 2003) is an example of an algorithm that treats censored measurements as uncensored measurements. If the duration of a transition from one state region to another exceeds some specified threshold, TTree terminates the transition and uses the measurement of the transition duration up to the point of termination.

Examples of reinforcement learning algorithms that ignore censored data can be found in the work by Munos (2000). He presents model-based and model-free algorithms for continuous-event, deterministic systems. He uses a finite element scheme to estimate a model, where the elements are points embedded in an Euclidean state space. When the agent passes through a neighbourhood of one of the elements, the agent uses the duration of the trajectory through the neighbourhood to estimate one of the model parameters. If the trajectory is interrupted, perhaps by a change in action, then the measurement is ignored.

If trajectory measurements are rarely censored, then treating censored measurements as uncensored measurements or ignoring censored measurements will not be disastrous. However, in *a priori* unknown domains with large action spaces, a significant portion of trajectories are likely to be interrupted, making proper estimation from censored data important. The main difficulty in handling censored data is assuming a parametric probability distribution over the measurements.

As this chapter explains, exploration is an important issue in modelling. A poor exploration strategy can prevent the agent from gaining the experience necessary to construct a useful plan. Section 5.3 discusses undirected and directed exploration strategies and presents a new directed strategy based on a heuristic suggested by Moore and Atkeson (1993). Experiments with this new strategy are presented later in Chapter 8.

In order to avoid random exploration, prior knowledge must be used to bias the agent towards good behaviour. As this chapter discusses, there are many ways in which to bias the learning process. One of the most successful approaches in the literature is the use of guided exploration where a teacher guides the reinforcement learning agent through a set of examples. For some problems, it only takes a single training episode to expose the agent to enough of the state space so that it no longer has to rely on random exploration. The experiments in Chapter 10 use guided exploration when evaluating AMPS.

Chapter 6

Distinction

This chapter explains how AMPS distinguishes states and actions. Following the introductory discussion of the first section, the second section discusses heuristics for determining which states or actions within a region to split. The third, fourth, and fifth sections explain how decision graphs and nearest neighbour approaches achieve separation. The sixth section explains value clipping, which is a necessary process when combining map revision and planning. The final section discusses the contributions of this chapter.

6.1 Introduction

In the absence of any prior knowledge about the dynamics of the world, the agent begins with the initial hypothesis that there is no reason to prefer executing one action over another. This initial hypothesis follows the law of parsimony as described by Newton in *Philosophiæ Naturalis Principia Mathematica* (1687),

> Causas rerum naturalium non plures admitti debere, quam quæ & veræ sint & earum phænomenis explicandis sufficiant.[1] (page 402)

As the agent accumulates experience in the world, it must revise its hypotheses to conform to its observations, in much the same way as a scientist. The cause-and-effect relationships are encoded in a model, and refining the model involves making distinctions in the state and action spaces.

To illustrate hypothesis revision, consider the robot football domain where the agent might encode the hypothesis *if I am close to a ball and I kick then I will*

[1] *We are to admit no more causes of natural things than such as are both true and sufficient to explain their appearances.*

score a goal. Further experience in the world might indicate that this hypothesis does not always hold and that the hypothesis should instead be *if I am close to a ball and I kick then I will score a goal half the time.* This hypothesis is valid and the agent is free to accept this nondeterminism as stochasticity inherent in the system. After all, scientists and economists frequently incorporate stochasticity in their models. However, it is likely to be beneficial to the agent to make distinctions between states or actions. For example, the agent might revise its hypothesis to *if I am close to a ball and facing the goal and I kick then I will score a goal* or *if I am close to a ball and kick in the direction of the goal then I will score a goal.* Deciding when to accept non-determinism, make perceptual distinctions, or make actional distinctions is key to successful modelling and, therefore, successful behaviour.

The distinctions that are important to incorporate in a model are dependent on the task. If the objective is to score as many goals as possible, making perceptual distinctions with respect to the goal is essential for satisfactory behaviour. However, if the objective of the agent is to simply keep the ball away from another player, then there is no need to make perceptual distinctions with respect to the goal. It is important to avoid introducing more distinctions than necessary for the task because too many distinctions impair generalisation. The modelling process in AMPS uses heuristics to decide how and when to make distinctions.

Making distinctions in the state and action spaces requires computation on the underlying representation. AMPS limits all processing of the underlying representation to a self-contained module for two reasons. The first reason is that it enables AMPS to be applied to a wide variety of domains with only the representation-dependent module requiring special engineering. The second reason is that *representation is everything.* The successful application of any learning algorithm depends strongly on the representation. A domain expert can engineer the representation-processing models in AMPS to leverage the representation. Engineering the module is not necessarily difficult or time consuming. The current implementation of AMPS includes tools and algorithms for handling a variety of representations (Sections 6.4 and 6.5).

This chapter proceeds as follows. The next section introduces the heuristics used by AMPS to decide when and how to introduce distinctions in the state and action spaces. The chapter continues with a presentation of two very different approaches for introducing perceptual and actional distinctions. The first

approach involves a decision graph where distinctions are made by tests that perform computation on the underlying representation of the states and actions. The second approach relies upon a distance metric for instance-based generalisation. Following this discussion, this chapter presents value clipping, which is necessary for efficient planning when interleaved with incremental map revision. The final section discusses and summarises the contributions of this chapter.

6.2 Heuristics

This section describes two kinds of splitting heuristics in AMPS, *value revision* and *failure revision*. These heuristics attempt to separate groups of observed trajectories that exhibit different behaviour. The heuristics determine the grouping of trajectories into disjoint sets $T_1, \ldots, T_n$. This grouping is presented to the $\textsc{Split}(T_1, \ldots, T_n)$ function, which attempts to distinguish these trajectories by introducing a representation-dependent distinction in the state or action space as Section 6.3 later describes.

Both the value-revision and failure-revision heuristics look at states and actions involved in *greedy transitions*. Given a policy π computed using the dynamic programming techniques of Chapter 8, a greedy transition from S is one that transitions to another region S' by actions in $\pi(S)$.

6.2.1 Value Revision

The objective of value revision is to ensure that all transitions resulting from a greedy action have approximately the same estimated value. Value revision attempts to separate state-action pairs in transitions with differing estimated value. The value of a transition from S to S' by A is

$$Q(S, A, S') \equiv \gamma(S, A, S')r(S, A, S') + \lambda(S, A, S')\rho(S, A, S') + \gamma(S, A, S')V(S').$$

Notice that

$$Q(S, A) = \sum_{S' \in \mathcal{S}} P(S' \mid S, A)Q(S, A, S').$$

At a region S and for actions in A, the variation of $Q(S, A, S')$ over different successor regions S' could be due to stochasticity or *aliasing*. Aliasing occurs when two states or actions that behave differently are grouped into the same region. If the variation of the transition values are significantly different, it

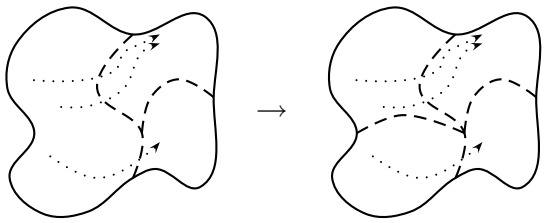


Figure 6.1: Value revision. On the left, two trajectories lead to a high-valued region, and one trajectory leads to a low-valued region. Because these trajectories are of different value, it is likely to be beneficial to split the source region as shown on the right.

might be beneficial to distinguish the states and actions involved in higher-valued transitions from those involved in lower-valued transitions. Figure 6.1 shows an example of how the state space might be split.

The agent must decide which trajectories should be kept together and which trajectories should be kept apart based only on their transition values. The problem reduces to *clustering*, a kind of unsupervised learning. There are many algorithms for grouping items into k clusters as surveyed by Xu and Wunsch (2005). Most clustering algorithms attempt to minimise some criterion function when assigning items to clusters. The k-means algorithm, for example, minimises the sum-squared error. The k-means algorithm is extremely efficient, running in $O(kn)$ time per iteration where n is the number of items to be clustered.

One of the difficulties of clustering is deciding how many clusters exist in the data. There have been many techniques proposed for determining the number of clusters (Milligan and Cooper, 1985). Many of the techniques involve incrementally increasing the number of clusters until there is a steep change in some criterion function. Everitt et al. (2001, Section 5.5) discuss such techniques further, including some more principled approaches for determining the number of clusters, but they are far from efficient.

The current implementation of AMPS simply computes the mean value of the transitions from a state-action region, and it splits the trajectories involved in transitions according to whether their values are above or below the mean. Figure 6.2 shows a cluster decision boundary at the mean of the transition values. Of course, this process produces only two clusters. Although the values plotted in the figure indicate the existence of only two clusters, this need not be the case. Splitting at the mean is generally a poor choice given the data from the point

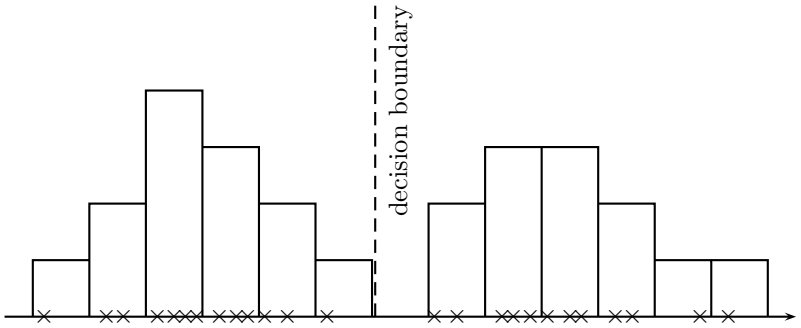


Figure 6.2: Transition values and their decision boundary. Plotted on the horizontal axis are the values of various transitions from a region. The histogram illustrates the distribution of the samples. This decision boundary indicated by the dashed line is at the mean of the samples.

of view of optimising some criterion function. However, suboptimal clustering does not seem to impact the actual performance of the agent since it is given opportunities to later refine the clusters.

Because it is not practical for an agent to split every state that needs splitting while interacting with the world, the agent prioritises the splits. The priority of a split is related to the variability of the transition values. In AMPS, the priority of splitting at a state region S is given by the variance of $Q(S, \pi(S), S')$. When the agent decides to split S, it will compute the mean transition value and call Split to separate the transitions whose values are no greater than the mean from those greater than the mean.

6.2.2 Failure Revision

Failure revision uses a signal to detect whether the continual application of an action is likely to result in progress. One way to detect failure is through sensory information from the environment. For example, in a wall-following task, a bump sensor can indicate that forward motion is not possible. Another way to detect failure is with a timeout. If the continual application of greedy actions from a state region does not result in a transition to a new state region within a prescribed amount of time, it is likely that aliasing is hindering progress and that adding perceptual or actional distinctions would be useful.

The Parti-Game algorithm (Moore and Atkeson, 1995) uses a similar failure signal to detect when the agent becomes "stuck" to revise its partition of the state

space. TTree (Uther, 2002) detects deterministic self-transitions, an indication of failure, to determine when to split the state space. It is important that the agent has a failure signal, otherwise the agent does not know whether a transition is simply taking a long time or whether it is really stuck and should take another course of action.

The priority of failure revision at a particular state region S is equal to the number of greedy trajectories resulting in failure divided by the total number of greedy trajectories resulting in either failure or a successful transition to another region or termination. When the agent revises S, it will split transitions resulting in success from those resulting in failure.

6.3 Trajectory Separation

Having identified which trajectories need to be separated (using the heuristics in the previous section), the agent must decide how to separate the trajectories. The procedure $\textsc{Split}(T_1, \ldots, T_n)$ may introduce a distinction in the state space or in the action space to separate the states or actions involved in the trajectories belonging to different sets. If the agent decides to split state region S, it will call $\textsc{Split}(S, S_1, \ldots, S_n)$, where S_k is a set consisting of the states in the trajectories of T_k. If the agent decides to split action region A, it will call $\textsc{Split}(A, A_1, \ldots, A_n)$, where A_k is a set consisting of the actions in the trajectories of T_k.

The agent must rely on a heuristic to decide whether to split in the state space or action space. Information gain (Shannon, 1948) may be used as an indicator of which way of splitting is best. It might be the case that both ways of splitting give the same information gain, in which case the agent might use some measure of generalisation error such as cross-validation (Lachenbruch and Mickey, 1968; Kohavi, 1995) or bootstrapping (Efron, 1979, 1983; Efron and Tibshirani, 1993). Some kinds of bootstrapping, such as the 0.632 Bootstrap, provide poor estimates of generalisation error for nearest neighbour classification (Jain et al., 1987). Weiss (1991) empirically shows that stratified twofold cross-validation is the best generalisation estimator for nearest neighbour in comparison to other cross-validation and bootstrap techniques. Computing cross-validation error can be expensive, so it may be advantageous to simply split in the state space by default when splitting in the action space does not perfectly split the trajectories.

Once the agent decides whether to split in the state or the action space,

it calls either SPLIT$(S, S_1, \ldots, S_n)$ or SPLIT$(A, A_1, \ldots, A_n)$ accordingly. The current implementation of AMPS includes two alternatives for supporting the SPLIT operations, one involving decision graphs and the other involving nearest neighbour metrics. The next two sections discuss these approaches in depth. To keep the discussion generic, these sections refer to the splitting of general "objects" instead of specifically states or actions. So, instead of describing SPLIT$(S, S_1, \ldots, S_n)$ and SPLIT$(A, A_1, \ldots, A_n)$ separately, these sections describe the generic SPLIT$(X, X_1, \ldots, X_n)$, which splits the sets of objects $X_1, \ldots, X_n$ belonging to region X.

The procedure SPLIT$(X, X_1, \ldots, X_n)$ performs what may be viewed as supervised learning. Objects in the set X_i may be thought of as training examples that belong to class i. The objective is to revise the map by dividing region X into a set of new regions such that the training examples belonging to different classes get mapped to different regions. As the next section discusses, the split may not be perfect and it is important to avoid overfitting the training data.

Both the decision graph approach and the nearest neighbour approach are suitable for general representations. An object might be represented, for example, as a binary bit string, a vector of real values, a set of logical sentences, or XML.[2] The decision graph and nearest neighbour approaches require computation over the underlying representation of the objects. The decision graph computes the outcome of a test based on the underlying representation, and nearest neighbour computes distances based on the underlying representation. The next two sections discuss ways of exploiting the structure inherent in the underlying representation.

6.4 Decision Graph Approach

One way to implement the map is with a decision graph. A decision graph can very quickly map a state or an action to a region through a series of tests. Figure 6.3 shows how a decision graph may partition both the state and action spaces. In general, as the previous section mentions, different data structures might partition the state and action spaces.

Introducing distinctions in a decision graph involves introducing a test at a leaf node. The agent wishes to introduce the highest quality test according to some measure. The success of the decision graph in splitting the objects depends

[2]The XML 1.1 specification is available at `http://www.w3.org/TR/xml11/`.

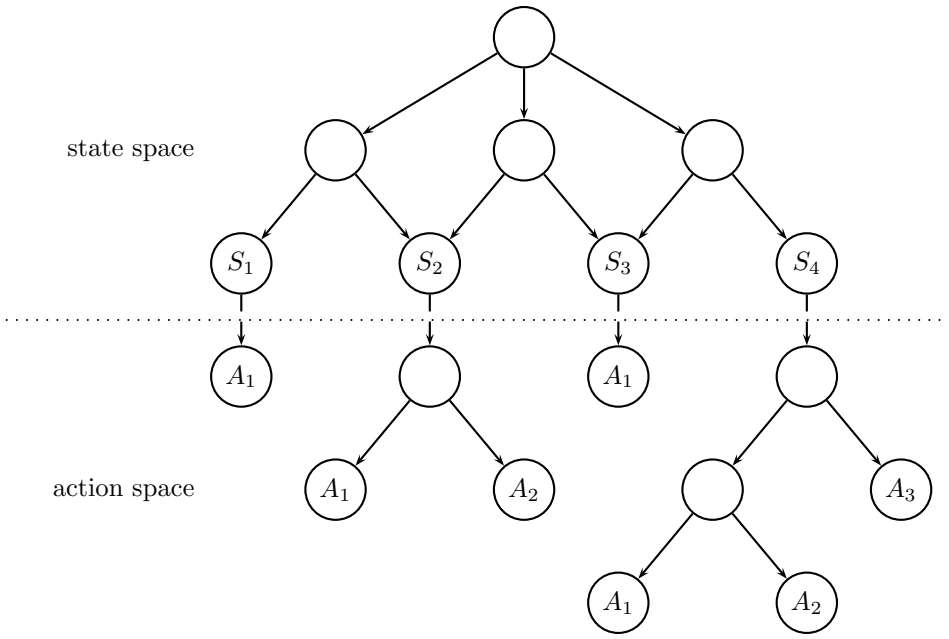


Figure 6.3: A decision graph partitioning the state and action spaces.

on the available tests. For some problem representations, the space of possible tests is infinite. For other representations, the space of tests is finite. It may not be possible, or even desirable, to split the objects perfectly. For efficiency and to avoid overfitting, the agent should introduce simple tests that are fast to compute.

This section uses $\hat{X}$ to represent the union of the sets $X_1, \ldots, X_n$. The procedure SPLIT$(X, X_1, \ldots, X_n)$ results in the redistribution of the objects in $\hat{X}$ into the sets $\hat{X}_1, \ldots, \hat{X}_m$. The remainder of this section discusses distinction quality measures and efficient ways to handle attribute-value representations, vector representations, and more general representations.

6.4.1 Distinction Quality Measures

Splitting $X_1, \ldots, X_n$ involves choosing a test that maximises some measure of quality. There are many different ways to measure the quality of a split. AMPS, like other tree-induction systems, defines the quality of a split in terms of the decrease in impurity, where impurity is a measure of the mixture of objects belonging to different categories. One such impurity measure borrowed from information

theory (Shannon, 1948) is *entropy impurity*, which is given by

$$i(X') = -\sum_{i=1}^{n} P(X_i \mid X') \lg P(X_i \mid X'),$$

where X' is some subset of X, the union of the sets $X_1, \ldots, X_n$. For any two sets X' and X'',

$$P(X' \mid X'') \equiv |X' \cap X''| \,/\, |X''| \,.$$

The decrease in impurity is the weighted impurity after the split subtracted from the impurity before the split,

$$i(X) - \sum_{j=1}^{m} P(\hat{X}_j \mid X) i(\hat{X}_j).$$

This quantity is also called the *information gain* (Shannon, 1948). If the information gain of a split is small, then it is not advantageous to introduce the distinction. An alternative to the information gain as a quality measure is the *gain ratio*, which is the information gain divided by

$$-\sum_{j=1}^{m} P(\hat{X}_j \mid X) \lg P(\hat{X}_j \mid X).$$

Quinlan (1986) suggests using the gain ratio so that the quality measure does not inherently favour splits with many outcomes. López de Mántaras (1991) suggests an alternative to the gain ratio that relies on a distance metric.

An alternative to entropy impurity is the *Gini impurity*, as used in CART (Breiman et al., 1984),

$$i(X') = \frac{1}{2}\left(1 - \sum_{i=1}^{n} P^2(X_i \mid X')\right).$$

The *minority impurity* is given by

$$i(X') = \sum_{i=1}^{n} P(X_i \mid X') - \max_{i \in \{1,\ldots,n\}} P(X_i \mid X').$$

The minority impurity is often used to produce the *sum minority* measure,

$$\sum_{j=1}^{m} i(\hat{X}_j).$$

Minimising the sum minority is equivalent to minimising the number of misclassified objects. The literature contains many other impurity and quality measures (see Murthy et al., 1994). AMPS uses information gain by default.

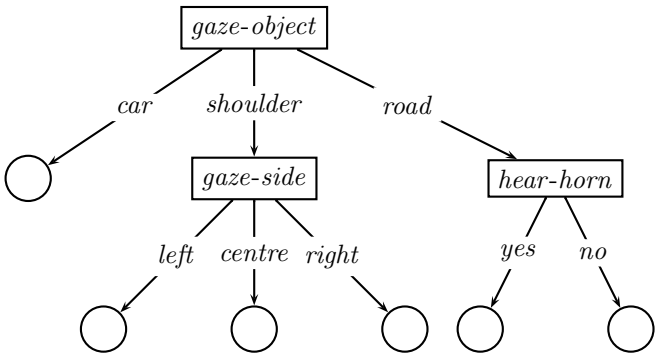


Figure 6.4: A decision graph for an attribute-value representation. Note that the *gaze-object* and *gaze-side* attribute tests have more than two outcomes.

6.4.2 Attribute-Value Representation

Attribute-value representations associate values to attributes. McCallum (1995) assumes an attribute-value representation for his UTree algorithm and tests his system on the New York driving task. Examples of attribute-value pairs to describe the system state include

$$(\textit{hear-horn}, \textit{no}), (\textit{gaze-distance}, \textit{far}), (\textit{gaze-object}, \textit{road}).$$

A single state is simply an assignment of values to all of the attributes. Assuming some ordering of the attributes, the state may be equivalently represented as a tuple of values. The domain of values for a particular component is finite.

Calling $\textsc{Split}(X, X_1, \ldots, X_n)$ on sets of objects associated with the region X involves choosing an attribute test that maximises the distinction quality. If the attribute has m different possible values, then the split partitions X into $X'_1, \ldots, X'_m$. Figure 6.4 illustrates a decision graph for an attribute-value representation.

6.4.3 Vector Representation

Attribute-value representations are often not appropriate for real-world applications. Vector representations are more general and allow for greater generalisation. Whereas there is no implied relationship between the nominal values assigned to an attribute in an attribute-value representation, a vector representation with an associated inner product defines relationships between objects

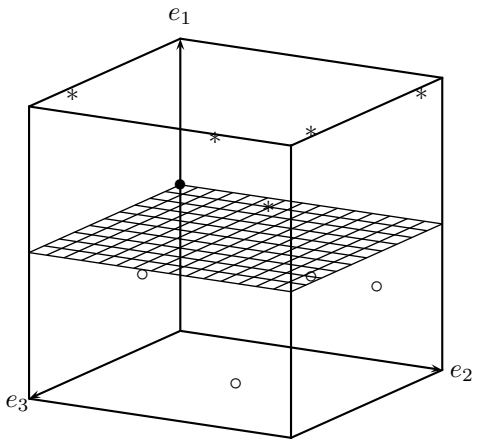


Figure 6.5: A basis-orthogonal decision boundary. The cut point on basis e_1 is indicated by ($\bullet$). Samples from class ($*$) and class ($\circ$) are indicated as points in the space.

including angles and distances. Supervised learning methods typically involve generalisation over objects embedded in a vector space.

A vector space H endowed with an inner product $\langle \cdot, \cdot \rangle$ is called an inner-product space.[3] In general, vector spaces may be over any field, but AMPS and most supervised learning methods assume that the vector space is over the reals and that the inner product of two objects in H is real. One example of a real inner-product space is Euclidean space with the standard dot product serving as the inner product. In Euclidean space, a vector is a tuple of real values. Continuous UTree (Uther, 2002), CART (Breiman et al., 1984), and many other algorithms assume that the state space may be embedded in Euclidean space.

The objects, i.e. the states or actions, may belong to any domain. Here, ϕ is a function that maps arbitrary objects to vectors in H. In other words, ϕ transforms the object space into a vector space. The agent may use a variety of approaches to introduce distinctions in this vector space.

Basis-Orthogonal Hyperplanes

Many of the standard decision tree induction algorithms, such as CART (Breiman et al., 1984), ID3 (Quinlan, 1986), and C4.5 (Quinlan, 1993), use *cut points* to split continuous attributes (see also Fayyad and Irani, 1992). This approach may

[3]For a review of linear algebra see the introductory text by Anton (2005).

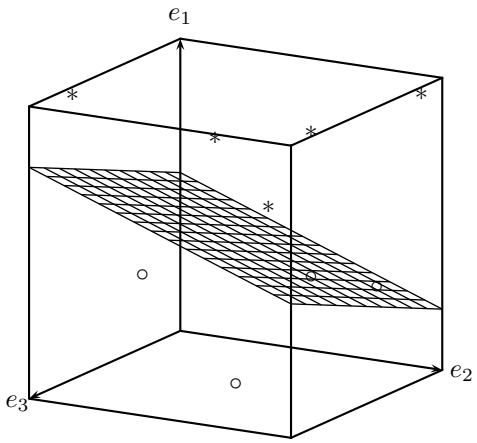


Figure 6.6: An oblique decision boundary. Samples from class ($*$) and class ($\circ$) are indicated as points in the space.

be generalised to any d-dimensional vector space H with bases $e_1, \ldots, e_d$. Introducing a cut point on some basis vector is equivalent to introducing a hyperplane orthogonal to that basis vector. The hyperplane results in separating the objects in $X_1, \ldots, X_n$ into the sets $\hat{X}_1$ and $\hat{X}_2$. Figure 6.5 shows an example of a basis-orthogonal decision boundary.

AMPS contains an efficient implementation for finding a basis-orthogonal hyperplane that maximises information gain. FIND-BEST-SPLIT$(e_k, \phi, X_1, \ldots, X_n)$ computes the optimal cut point along the basis e_k. To compute the best split, the agent calls FIND-BEST-SPLIT on each basis vector and cuts along the one that results in minimal weighted entropy. Computing the best cut point on a basis can be done with a single pass through the objects (Algorithm 1). The algorithm considers cut points between objects projected on e_k and computes the entropy using counts that it maintains as it passes through the list of values. If there are a total of m objects belonging to n classes, the algorithm runs in $O(mn + m \log m)$ time when using an $O(m \log m)$ sorting algorithm (Cormen et al., 2001).

Oblique Hyperplanes

Restricting the decision boundaries to basis-orthogonal hyperplanes limits how well they can separate the objects. Although allowing oblique hyperplanes can increase the information gain, the algorithms for choosing arbitrary hyperplanes have greater complexity.

Algorithm 1 FIND-BEST-SPLIT$(e_k, \phi, X_1, \ldots, X_n)$

$L \leftarrow \bigcup_{i=1}^{n} \{(\langle e_k, \phi(x)\rangle, i) \mid x \in X_i\}$
sort L in ascending order according to its first component
last $\leftarrow$ NIL
best-split $\leftarrow$ NIL
min-entropy $\leftarrow \infty$
seen $\leftarrow$ an array of length n with all elements set to 0
$n_1 \leftarrow 0$
$n_2 \leftarrow \sum_{i=1}^{n} |X_i|$
for all $(c, j) \in L$ **do**
  if *last* $\neq$ NIL and $c \neq$ *last* **then**
    $entropy = \frac{-n_1}{n_1+n_2} \sum_{i=1}^{n} \frac{seen[i]}{n_1} \lg \frac{seen[i]}{n_1} + \frac{-n_2}{n_1+n_2} \sum_{i=1}^{n} \frac{|X_i|-seen[i]}{n_2} \lg \frac{|X_i|-seen[i]}{n_2}$
    if *entropy* $<$ *min-entropy* **then**
      min-entropy $\leftarrow$ *entropy*
      best-split $\leftarrow (last + c)/2$
  $seen[j] \leftarrow seen[j] + 1$
  $n_1 \leftarrow n_1 + 1$
  $n_2 \leftarrow n_2 - 1$
  last $\leftarrow c$
return (*best-split*, *min-entropy*)

Oblique decision trees use arbitrary hyperplanes to separate objects into two categories (Figure 6.6). A vector θ in H and scalar c in $\mathbb{R}$ uniquely define a separating hyperplane. Any point p on the hyperplane satisfies $\langle \theta, p \rangle = c$. The hyperplane splits the vector space into two subspaces. An object x in the left subspace, satisfying the linear test $\langle \theta, \phi(x) \rangle \leq c$, belongs to one class, and an object x' in the right subspace belongs to the other class. Sometimes decision trees that use these linear tests are called *perceptron trees* (e.g., Utgoff, 1989; Bennett et al., 2000). Because the tests involve multiple variables, an oblique tree is a kind of *multivariate* decision tree (Brodley and Utgoff, 1995).

In general, finding the best hyperplane according to some metric is not practical. As proven by Heath (1992, Appendix C), even minimising the sum-minority metric is NP complete (see also Höffgen et al., 1995; Ben-David et al., 2003). Hence, a number of algorithms employ heuristic search methods. CART (Breiman et al., 1984, Section 5.2) uses a deterministic heuristic search procedure for the vector θ and the scalar c that approximately maximises the information gain. Heath et al. (1993) use simulated annealing to find the hyperplane that approximately maximises the quality according to the sum-minority metric. The OC1 tree-induction system (Murthy et al., 1994) uses local heuristic search to find a locally optimal split and then perturbs the hyperplane.

The quality measures in Section 6.4.1 do not utilise the structure inherent in the vector space. It can be advantageous to use the inner product to define the measure of quality. For example, one might wish to minimise the sum of the distances of misclassified objects to the separating hyperplane. Interestingly, there are efficient methods that optimise over this measure of quality, even though optimising the sum-minority measure is NP complete. Duda et al. (2000, Chapter 5) survey a variety of methods for choosing decision boundaries that maximise criteria involving inner products.

Some applications involve vector spaces of high or infinite dimensionality, making transforming the objects to vectors in H and computing their inner product impractical. Instead of computing $\langle \phi(x), \phi(x') \rangle$ directly, it may be possible to define a *kernel function* $k(x, x')$ to be used in its place. Although $k(x, x') = \langle \phi(x), \phi(x') \rangle$, the kernel function may be more efficient to compute. Many classification algorithms, such as *support vector machines* (see Cristianini and Shawe-Taylor, 2000; Herbrich, 2002), rely solely on the inner product, and their performance can be greatly enhanced with an appropriate choice of kernel

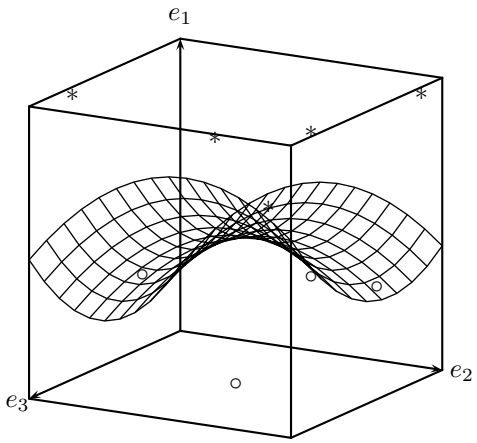


Figure 6.7: A nonlinear decision boundary. Samples from class ($*$) and class ($\circ$) are indicated as points in the space.

function. The literature contains a number of kernel functions for a variety of representations including graphs, sets, unstructured text, and structured data (Schölkopf and Smola, 2000; Shawe-Taylor and Cristianini, 2004).

Oblique decision tree induction continues to be an active area of research. Recent research includes the application of evolutionary algorithms (Krętowski, 2004) and minimum message length heuristics (Tan and Dowe, 2004).

Nonlinear Decision Boundaries

Nonlinear decision boundaries can better separate objects belonging to different categories, although they have a greater risk of overfitting the data. Figure 6.7 illustrates a nonlinear decision boundary in three dimensions. One way to achieve nonlinear decision boundaries is by transforming H to a higher dimensional vector space and creating a linear decision boundary.

Another way to achieve nonlinear decision boundaries is by using a neural network with one or more hidden layers. Guo and Gelfand (1992) describe how to use neural networks as tests in decision tree induction. They modify the back-propagation algorithm to reduce the Gini impurity. Yıldız and Alpaydın (2001) use the 5×2 cv F test (Alpaydın, 1999) to decide whether to introduce univariate (i.e., basis-orthogonal), linear, or nonlinear tests. They call their trees *omnivariate* decision trees.

6.4.4 Generating Tests

For some domains, especially relational domains, it is not natural or desirable to transform the object space into a vector space and define an inner product. AMPS provides an alternative way to generate tests for a decision graph. The system synthesises tests from a specification of typed predicates and functions that perform operations on the underlying representation of the object space. AMPS assembles the test that maximises the information gain.

The current implementation of AMPS accepts an XML specification of a hierarchical type system. The first part of the file specifies the names of the types and "is-a" relationships between the types. The second part of the file specifies the functions. Each function has a name, a return type, a set of typed parameters, and a location of the function. The third part of the file specifies the predicates, including their name, typed parameters, and location.

Presently, the string in the XML file denoting the location of a function or predicate is a fully-qualified class name of the Java implementation. However, it is possible to extend the implementation of AMPS to support functions implemented in other languages in remote locations or calls to XML Web Services.[4] The AMPS package includes a variety of general-purpose types, functions, and predicates, but in principle AMPS can utilise a distributed library of types, functions, and predicates.

The types do not simply denote how the underlying data is represented; the types convey semantics. Although two types might share the same representation encoded as an XML Schema[5], they may have different semantics. A sequence of integers, for example, might represent an unordered set of integers, an ordered set of integers, or a binary tree with integers associated with the nodes. When introducing distinctions between objects, it is important to be faithful to the semantics. In order to differentiate between types with different semantics, one might assign a Unique Resource Identifier (URI) to the type.[6]

AMPS constructs all of the well-formed grounded predicates within some depth limit in memory. A maximum depth must be imposed because it is possible for functions to be infinitely nested. For example, the function f might take a

[4]For further details on XML Web Services, see `http://www.w3.org/2002/ws/`.

[5]See `http://www.w3.org/XML/Schema`.

[6]URIs are an important concept in the emerging semantic web. The URI specification is available at `http://www.w3.org/Addressing/`. The URI of a type might also be a location of a document that explains the semantics of the type.

parameter of type τ and return a value of type τ, allowing itself to be infinitely nested. When AMPS introduces a test into a decision tree, it simply points to the grounded-predicate already in memory.

Although a properly designed type system can greatly restrict the number of possible grounded predicates, it is likely that AMPS produces tautologous, inconsistent, or redundant grounded predicates with respect to a set of axioms Ψ. Such predicates should be removed from consideration. A grounded predicate ψ is tautologous when $\Psi \models \psi$ and is inconsistent when $\Psi \models \neg\psi$. Such grounded predicates are not useful in distinguishing objects. If ψ and ψ' are grounded predicates, and $\Psi \models (\psi \leftrightarrow \psi') \vee (\psi \leftrightarrow \neg\psi')$ then ψ is redundant with ψ' and should be removed from consideration.

AMPS uses the JTP automated reasoning system (Fikes et al., 2003) to prove whether a grounded predicate can be pruned from consideration. In order to prove whether a grounded predicate is useful, one must specify a set of axioms over the functions and predicates. For example, in the Taxi World domain, some useful axioms include $X = X$, $\neg\, north(X, X)$, and $east(X, Y) \leftrightarrow west(Y, X)$, with X and Y as universally quantified variables. The axioms are specified in the standardised Knowledge Interchange Format (Genesereth and Fikes, 1992).

Pruning unnecessary grounded predicates with a theorem prover is not trivial, requiring minutes to compute. Fortunately, this pruning only needs to occur once, before the agent commences interaction with the world. AMPS also includes the functionality to serialise to a file the set of pruned ground predicates in memory. The compiled file can later be restored to memory quickly, without having to use the theorem prover.

When the agent needs to decide upon a grounded predicate that best splits $X_1, \ldots, X_n$, it iterates through the possible grounded predicates and chooses the one that maximises the information gain or some alternative quality measure. Assuming that evaluating grounded predicates requires constant time, the computational complexity of choosing the best grounded predicate is $O(mn + \ell n)$, where m is the number of objects in $X_1, \ldots, X_n$ and ℓ is the number of grounded predicates.

AMPS stores the objects using their natural representation. However, this is not strictly necessary from the point of view of adding distinctions. Suppose objects are naturally represented as sets of lists. Instead of storing the object using some representation denoting a set of lists, the agent could simply store the

object as a binary bit string corresponding to the truth values of the grounded predicates when evaluated on that object. When the agent must decide which test to use, it simply chooses the grounded predicate corresponding to the bit that maximises information gain. Such an approach is analogous to defining a function ϕ that maps the objects to a binary bit string and then using a basis-orthogonal splitting scheme, as discussed earlier. The cut points will be at 0, separating the false predicates at -1 from the true predicates at $+1$.

6.5 Nearest Neighbour Approach

The previous section explains various ways to add distinctions and achieve generalisation with decision graphs. This section discusses a completely different approach supported by AMPS that involves nearest neighbour generalisation (Cover and Hart, 1967). Nearest neighbour is a kind of instance-based learning algorithm (Aha et al., 1991) where labelled instances are generalised to unlabelled objects by means of a distance metric. In AMPS, the labels correspond to regions of the state or action space. When presented with a novel object, the nearest neighbour rule predicts its label to be the same as its closest neighbour. The objects may have any representation so long as one can define a suitable distance metric.

A *metric* is a nonnegative, symmetric function $d(x, x')$ that denotes the distance between the object x and object x'. Metrics satisfy $d(x, x) = 0$ and the *triangle inequality*

$$d(x, x') + d(x', x'') \geq d(x, x'').$$

If $d(x, x') = 0$ does not necessarily imply $x = x'$, then $d(x, x')$ is called a *pseudometric*. The discussion in this section about metrics also applies to pseudometrics. A set of objects endowed with a metric is called a *metric space*.

As the agent interacts with the world, it records the information from its trajectory, including the states observed and the actions taken. When the agent records a state or action, it associates with the object a label designating its region. The object receives the label of its closest neighbour according to some specified metric. This label is actually a reference to a linked list in memory consisting of all the objects previously experienced in the same region (Figure 6.8). After assigning the label to the object, AMPS updates the list by adding the new object. When the agent records its first object, the object will not have

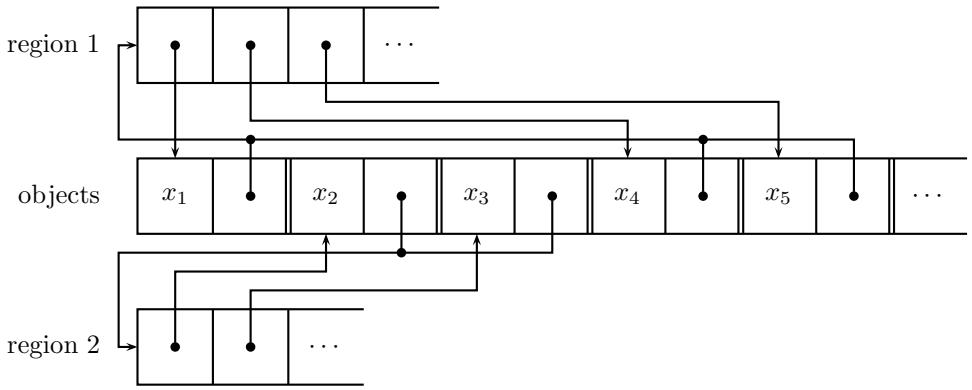


Figure 6.8: The nearest neighbour data structure. A pointer associates each object with its region. A region consists of a list of references to objects in the region. In the illustration, x_1, x_4, and x_5 belong to region 1, and x_2 and x_3 belong to region 2. This diagram is a simplification and does not show the additional information associated with objects and regions.

neighbours, so the agent associates with the object a new linked list consisting only of the object.

The procedure SPLIT$(X, X_1, \ldots, X_n)$ revises the regions so that the objects in the sets $X_1, \ldots, X_n$ get mapped to different regions and the remaining objects in X get mapped to the region of its closest neighbour (Algorithm 2). The process begins by iterating through each set X_i, first creating a new region and then moving all the objects in X_i from the old region X to the new region. After the iteration is complete, the process adds each remaining object in region X to the region of its closest neighbour in the sets $X_1, \ldots, X_n$.

Algorithm 2 SPLIT$(X, X_1, \ldots, X_n)$

for $i = 1$ to n **do**
  create a new region X'
  move the objects in X_i from X to X'
for all remaining objects x in X **do**
  add x to the region of its closest neighbour in the sets $X_1, \ldots, X_n$

Planning (Chapter 8) occurs over regions of the state space, not individual states. The learned policy maps state regions to action regions. When the agent encounters a new state, the agent must compute its region. The decision graph approach computes the region through a series of tests, but the nearest neighbour

approach computes the region by searching for the closest labelled instance. Because the agent must compute the nearest neighbour at least as frequently as it samples the environment, it is important that the nearest neighbour calculations are efficient.

A naïve implementation of nearest neighbour involves a full search through the set of instances. More efficient implementations utilise some form of indexing data structure. Many different applications require computing the nearest neighbours of a query point, and so the literature is full of indexing methods for efficient nearest neighbour calculation. Efficient nearest neighbour computation arises in a number of areas outwith machine learning, including information theory involving vector quantisation (Gray and Neuhoff, 1998) and databases involving similarity search (Papadopoulos and Manolopoulos, 2005).

The literature contains a wide variety of indexing methods for general metric spaces (e.g., Kalantari and McDonald, 1983; Yianilos, 1993; Ciaccia and Patella, 2002). Chávez et al. (2001) and Hjaltason and Samet (2003) survey several approaches and compare their efficiency. It should be emphasised that these indexing systems work with any metric space, regardless of distance measure or object set. Ramon (2002, Chapter 3) discusses a wide range of distance metrics in depth, including metrics over sets, trees, strings, and first-order logic objects. Wilson and Martinez (1997) describe other distance metrics. Although one might be able to define a distance metric for a space, there is no guarantee that the metric will provide the desired generalisation.

Inner-product spaces (see Section 6.4) have a naturally-defined distance metric. If the object space can be mapped by a function ϕ to a vector space H with inner product $\langle \cdot, \cdot \rangle$, then the distance between objects x and x' is

$$\begin{aligned} d(x, x') &= \sqrt{\langle \phi(x) - \phi(x'), \phi(x) - \phi(x') \rangle} \\ &= \sqrt{\langle \phi(x), \phi(x) \rangle + \langle \phi(x'), \phi(x') \rangle - 2 \langle \phi(x), \phi(x') \rangle} \\ &= \sqrt{k(x, x) + k(x', x') - 2k(x, x')}, \end{aligned}$$

where $k(\cdot, \cdot)$ is the kernel. Hence, any object space with a properly defined kernel may be treated as a metric space. As the previous section notes, kernel evaluations can be fast even if the dimensionality of the object space is large.

If the object space is assumed to be some Euclidean subspace, a common indexing approach involves data structures called k-d trees (Bentley, 1975; Friedman et al., 1977). The randomly-generated SMDPs in the experiments of Chap-

ter 8 use an implementation of k-d trees following the description of Moore (1990, Chapter 6). The k-d tree indexing method is extremely efficient and has been used in a variety of applications. For example, adaptations of the basic algorithm have been recently applied to protein comparison (Shyu et al., 2004) and searching large-scale medical diagnostic image databases (Scott and Shyu, 2003).

Besides indexing, another way to improve nearest neighbour classification efficiency is by pruning stored instances. Of course, less time is required to determine the nearest neighbour of a query point when there are fewer possibilities. Hart (1968) proposes an algorithm for removing instances without significantly impairing generalisation, and Gates (1972) suggests improvements. AMPS does not take such an approach to improve classification efficiency for two reasons. The first reason is that the pruning algorithms involve iterating through all of the stored objects multiple times, which is not practical for the agent to do regularly. The second reason is that the stored instances need to be kept in memory to allow the estimation of a system model (Chapter 5).

6.6 Value Clipping

This section introduces value clipping as used in AMPS. The necessity for value clipping arises due to the interleaving of incremental modelling and planning. Value clipping results in improved estimates of the value function by clipping the value function at state regions where the estimated value is easily recognised to be too high or too low. This process is especially useful in identifying situations where the agent should take exploratory moves, as this section explains.

The value-clipping algorithm in this section assumes that the agent receives zero reward except when terminating, i.e. when it transitions to the synthetic terminal region S_0 (see Section 5.4.2). The value of a state region S can be no greater than the maximum value of $r(S', A, S_0)$ over all action regions A from all state regions S' reachable from S. Similarly, the value of a state region S can be no less than the minimum value of $r(S', A, S_0)$ over all action regions A from all state regions S' reachable from S. In other words,

$$V(S) \leq \max_{S' \in \textsc{Reach}(S), A \in \mathcal{A}(S')} r(S', A, S_0) \tag{6.1}$$

$$V(S) \geq \min_{S' \in \textsc{Reach}(S), A \in \mathcal{A}(S')} r(S', A, S_0), \tag{6.2}$$

where $\textsc{Reach}(S)$ is the set of all state regions reachable from S in the model.

The value-clipping algorithm identifies when either Equation 6.1 or Equation 6.2 is violated by the estimated value function and clips the values appropriately. The value-clipping algorithm is composed of two separate algorithms. One algorithm clips from above and the other algorithm clips from below. Since the algorithm that clips from below is almost identical to the one that clips from above, this section only discusses clipping from above. The algorithm for clipping from above (Algorithm 3) involves calling the recursive helper procedure VALUE-CLIP-MAX-HELPER (Algorithm 4) until all of the state regions have been marked as being clipped. In Algorithm 4, $\text{pred}(S)$ is the set of all predecessors of S in the model.

Algorithm 3 VALUE-CLIP-MAX

```
create a list L consisting of the elements in 𝒮
sort L according to max_{A∈𝒜(S)} r(S, A, S_0)
for all regions S ∈ L do
  if S is not marked then
    VALUE-CLIP-MAX-HELPER(S, max_{A∈𝒜(S)} r(S, A, S_0))
```

Algorithm 4 VALUE-CLIP-MAX-HELPER(S, v)

```
mark S
if V(S) > v then
  V(S) ← v
for all S' ∈ pred(S) do
  if S' is not marked then
    VALUE-CLIP-MAX-HELPER(S', v)
```

For an analysis of the time complexity of this algorithm, let $n = |\mathcal{S}|$ and $m = |\mathcal{A}|$. The computation of $\max_{A \in \mathcal{A}(S)} r(S, A, S_0)$ for each S is $O(mn)$, and the sorting of the regions according to these values is $O(n \log n)$ in the worst case. When the algorithm clips and marks a region, it must inspect all of the predecessors of the region to see whether they have been clipped and marked. Because the algorithm terminates after marking all n regions, the time complexity of clipping the values is linear in ℓ, where

$$\ell = \sum_{S \in \mathcal{S}} |\text{pred}(S)| \,.$$

Hence, the total time complexity is $O(mn + n \log n + \ell)$. A proof of correctness of Algorithm 3 is not difficult.

The value-clipping algorithm is particularly useful when the value at a region gets clipped to zero, indicating that the model does not contain information on a satisfactory way to proceed to a goal. In such a situation, it is beneficial for the agent to explore. In this discussion, goal regions refer to regions where there is a non-zero probability of transitioning to S_0 with positive expected reward for some action. Nonterminal regions are regions where the probability of transitioning to S_0 is zero. A region S is alienated from another region S' if there is zero probability of ever reaching S' from S, regardless of the policy the agent follows.

When integrating incremental planning and modelling, it is common for an added distinction in the state space to alienate a nonterminal region from all goal regions. Assuming that reward is always nonnegative, the alienated region should have zero value. However, if the planning process assigned positive value to the region before it was alienated, it can take many iterations of prioritised value iteration for the value to go to zero.

Figure 6.9 provides a concrete example of when value clipping is useful. In this example, S_1 is a goal region, and S_2 and S_3 are nonterminal regions. As shown in the figure, neither of the nonterminal regions are alienated from the goal region. With the estimated transition probabilities and reward along with some set continuous-time discount factor β, the planning process estimates the value of S_1 to be 0.9, the value of S_2 to be 0.8, and the value of S_3 to be 0.7. As shown in the diagram, the map-revision process splits S_1 into the nonterminal region S_1' and goal region S_1''. The only paths exist between nonterminal regions. The planning process can update the value of the goal region to its true value in a single iteration, but it requires many iterations for the values of the nonterminal regions to converge approximately to their true values. Clearly, the nonterminal regions have zero value because they are alienated from the goal region. Before the planning process begins propagating value between S_1', S_2, and S_3, the values are set to 0.0, 0.8, and 0.7 respectively. Irrespective of the order these regions are updated by value iteration, their values will take a long time to converge within some small threshold of zero, especially if γ, as defined in Equation 3.2, is close to unity for the transitions. Instead of waiting for prioritised value iteration to correctly compute the value function, the agent should use value clipping to immediately set the values of S_1', S_2, and S_3 to zero.

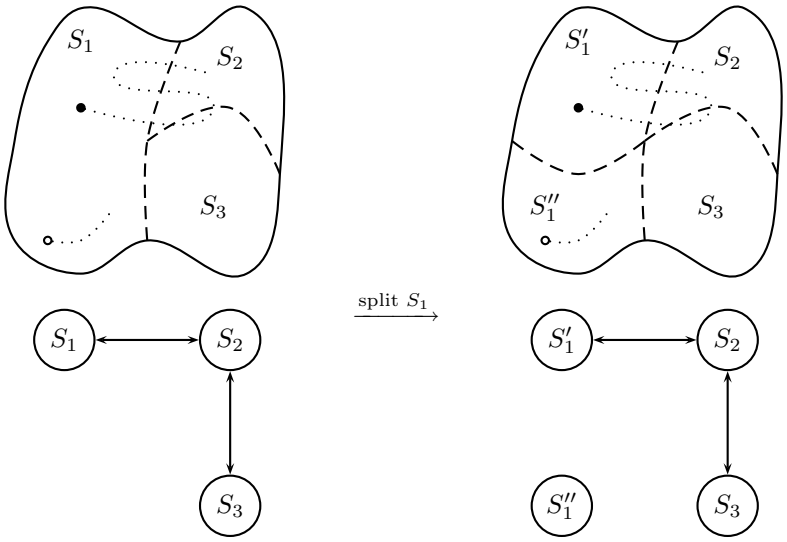


Figure 6.9: An example of when value clipping is useful. Originally the state space contains three regions, S_1, S_2, and S_3. The top-left diagram illustrates two trajectories in these regions. One trajectory terminates at a zero-reward state ($\bullet$), and another trajectory terminates at a unit-reward state ($\circ$). Below this illustration is a graph representing the observed connectivity between regions. The right side of the figure shows the resulting partition after the revision process splits S_1. Notice that there does not exist a path from S_1', S_2, or S_3 to a region with non-zero reward.

6.7 Discussion

The purpose of this chapter has been to discuss the theoretical and practical issues involved in introducing perceptual and actional distinctions. When the agent commences its exploration without prior knowledge of the system dynamics, its model, by the law of parsimony, assumes that all actions have the same effect from all states. After accumulating experience, the agent will likely discover that its model is overly simplistic and that it would be advantageous to adapt its model. An agent must be able to incrementally refine its model of the world in response to its incoming stream of experience. This section highlights the contributions this chapter makes in addressing the problem of deciding how and when to introduce distinctions in the state and action spaces.

If the agent observes that the world dynamics do not correspond to its internal model, it must decide whether the difference is due to stochasticity inherent in the system or aliasing. The agent must determine whether adding perceptual or actional distinctions will enhance its ability to accumulate reward. If the agent decides that it should introduce distinctions to reduce aliasing, it then needs to decide whether the distinction is necessary in the state space or the action space. In addition, the agent must decide exactly how to introduce this distinction.

The second section of this chapter offers two complementary heuristics for deciding how and when to introduce distinctions. The value-revision heuristic attempts to separate trajectories involved in transitions with significantly different values. The failure-revision heuristic attempts to separate trajectories that have encountered failure from those that have encountered success. Separating trajectories involves introducing distinctions in the state or action space. The third section presents approaches for choosing the space in which to introduce distinctions. These approaches involve principled heuristics based on information gain, cross validation, and bootstrapping. Previous tree-based generalisation systems for reinforcement learning (Chapter 4) only introduce distinctions in the state space and do not consider the issue of choosing whether to introduce perceptual distinctions or actional distinctions.

The approach AMPS takes in introducing distinctions is markedly different from other abstraction methods for reinforcement learning. The G, UTree, Continuous UTree, TTree, and TG algorithms (Section 4.2) all consider every possible way of introducing a split and then introduce the most statistically significant

split. AMPS, in contrast, decides first which samples it needs to split based only on the model and then uses a classifier to split the samples. Because AMPS avoids having to compute the statistical significance of every possible distinction, AMPS can introduce useful distinctions with far less computation. Not having to enumerate through every possible distinction also allows AMPS to select distinctions from sets too large to enumerate in practice.

The current implementation of AMPS includes two approaches to partitioning the state and action spaces. The first approach uses a decision graph. When one of the splitting heuristics (Section 6.2) indicates that some sets of states or actions need to be separated, the system searches for a test that maximises some quality measure. All of the quality measures in Section 6.4.1 have been well studied in the literature, especially information gain. A number of decision tree induction algorithms use information gain as a heuristic to decide which attribute to split, including CART (Breiman et al., 1984), ID3 (Quinlan, 1986), and C4.5 (Quinlan, 1993). The other decision tree based abstraction algorithms for reinforcement tasks (Section 4.2) do not use information gain as a metric because they do not treat the problem of splitting as a supervised learning problem.

This chapter presents ways of introducing splits in vector spaces as well as more general spaces. Because AMPS must introduce splits quickly, especially during the early stages of exploration, it is important that the splitting mechanism be efficient. Given sets of examples to split, it must quickly choose the perceptual or actional distinction that maximises the quality measure. This chapter introduces an algorithm for finding the best basis-orthogonal hyperplane according to information gain. This algorithm is used in AMPS for domains such as Race Track World.

Although basis-orthogonal hyperplanes provide sufficient separation for many domains, for other domains it might be desirable to introduce oblique hyperplanes. As this chapter explains, the actual representation of the state or action space need not be represented as a vector space so long as there is an appropriate kernel associated with the space. Kernel functions are available for a wide variety of representations due to the recent popularity of kernel methods for supervised learning.

This chapter describes a tool for generating a collection of tests for more general spaces. The current implementation of AMPS supports the automatic generation of tests based on a set of hierarchically typed functions. The type

hierarchy as well as the return types, parameter types, and references to the implementation of the functions may be specified in an XML file. The implemented system includes functionality for pruning redundant tests from consideration according to domain axioms in first-order logic. AMPS can do the pruning offline and serialise the tests to a file that may later be restored before commencing interaction with the world.

The nearest neighbour approach is an alternative to the decision graph approach for partitioning the state and action space. Computing the region to which a query point belongs requires determining which stored instance is closest to the query point according to some distance metric. Many different distance metrics exist for a wide variety of representations, and any kernel function can be adapted to serve as a distance metric. To make nearest neighbour calculations efficient, it is necessary to use some indexing approach, such as k-d tree indexing.

There have been other algorithms that utilise the nearest neighbour approach. Sections 2.3 and 4.3 in earlier chapters describe applications of nearest neighbour generalisation to supervised learning and reinforcement learning. However, the way in which AMPS uses nearest neighbour generalisation is different from any previous approach. When other model-based reinforcement learning algorithms use nearest neighbour for generalisation (e.g., Munos and Bourgine, 1998; Munos, 2000), each stored instance represents a different region in the abstract SMDP. In AMPS, however, multiple stored instances may belong to the same region. The approach used by AMPS provides for greater generalisation from limited experience. Although there are examples of model-free algorithms performing nearest neighbour actional generalisation (e.g., Smith, 2002a,b; Smart and Kaelbling, 2000; Smart, 2002), AMPS appears to be the first model-based reinforcement learning algorithm to allow nearest neighbour generalisation in the action space.

The previous section introduces value clipping. The necessity of value clipping was unexpectedly revealed during the early development and testing of AMPS. Although dynamic programming is known to provide estimates of the value function that converge to their true values in the limit, value clipping is necessary for fast convergence when regions of the state space are incrementally split as the agent accumulates experience. Value clipping involves clipping estimates of the value function at regions whose value estimates are easily determined to be too high or too low. This process is especially useful in determining when there is no known path from the current region to a goal region, indicating that the agent

should explore. Other reinforcement learning algorithms in the literature do not use processes analogous to value clipping, although some algorithms, for example UTree (McCallum, 1995, 1996a), would stand to benefit.

Perceptual and actional distinctions introduced by the agent early in its interaction with the world may later prove to be unnecessary. Hence, in addition to a process that increases the complexity of the model, there must be a process that simplifies the model when additional data indicates that simplification is warranted. The next chapter explains how model simplification is performed in AMPS.

Chapter 7

Simplification

This chapter discusses model simplification in AMPS. There are two kinds of simplification that are useful when constructing a model of the world from experience. The first kind of simplification involves merging regions of the state and action spaces. AMPS includes methods explained in this chapter for deciding exactly how and when to merge state and action regions. The second kind of simplification involves revising the representation of the mapping from states to state regions and actions to action regions. This chapter describes several methods for simplifying the mappings supported by decision graphs. The final section discusses the contributions of this chapter.

7.1 Introduction

The objective of science is to construct models from experience to explain natural processes. A multitude of models can explain the same set of observations, but, as the previous chapter mentions in its introduction, the law of parsimony is employed by the sciences to prefer simpler models over those of greater complexity. The model of a scientist, and indeed the model of a reinforcement learning agent, should be as simple as possible, but no simpler. Einstein (1934) once remarked

> It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience. (page 165)

This quotation may be interpreted from the perspective of an agent situated in an unknown world with the objective of maximising its utility. The "basic

elements" are the hypotheses of the agent about the system dynamics. In AMPS, these hypotheses are encoded as stochastic transitions in a graph, where the nodes of the graph correspond to regions of the state space. As Einstein states, these elements should be "as simple and as few as possible." In other words, the structure of the graph should be simple, and the nodes in the graph should be few. However, the simplicity of the graph should not be at the expense of "adequate representation." The model must account for all of its observations, and the model should be sufficient to allow the agent to accomplish its objective of maximising its utility. The principles encompassed in the quotation above, although intended for scientists, serve as a basis for AMPS.

AMPS is unique from other abstraction approaches (Section 4.2) in that it attempts to both simplify and refine its model. The other standard abstraction approaches only introduce perceptual distinctions when experience indicates that the current model is not sufficient to explain the observations. Consequently, the models monotonically increase in complexity as the agent acquires experience. However, data collected later by the agent might indicate that the model is unnecessarily complex and should be simplified. The process of evidence-based hypothesis revision is essential to the scientific method. As data is acquired in science, theories go through stages of both complexification and simplification (see Nye, 2002). The theories held by agents learning to behave in unknown environments ought to go through a similar process.

Complexification of the system model in AMPS involves splitting regions of the state and action space. Simplification involves merging regions. AMPS is more aggressive than the other abstraction algorithms, such as the G Algorithm (Chapman and Kaelbling, 1991) and UTree (McCallum, 1995, 1996a), in that it does not wait for sufficient data to prove statistical significance before performing a split. AMPS may introduce many more irrelevant distinctions in the state and action spaces than the other methods, but it does so with the idea that the model will be simplified later if further data indicates that it is useful to do so. The next section explains how AMPS goes about simplifying the model.

If the map in AMPS is supported by a decision graph (Section 6.4), splitting and merging regions can result in unnecessarily complex decision graphs. Although unnecessary complexity in the representation of the decision graph does not affect the quality of the model, it is important to expend some effort towards the simplification of the decision graph. Overly complex decision graphs can make

mapping states and actions inefficient. Since states and actions are mapped with each sample and decision, it is important that the mapping is efficient. Complex decision graphs also waste memory and are difficult for humans to understand and debug. Section 7.3 explains how AMPS simplifies decision graphs online. Note that the nearest neighbour approach to partitioning regions is not susceptible to the same kind of complexity issues arising from splitting and merging regions.

7.2 Model Simplification

This section explains how AMPS incrementally simplifies the structure of the model. Simplification involves either merging regions of the state space or merging regions of the action space. The first part of this discussion assumes that the action space is small and does not require generalisation and explains generalisation in the state space. The second part extends the discussion to problems with large action spaces that require generalisation.

Suppose for the moment that the model is deterministic and that the agent possesses some optimal policy π. Assume also that the action space is small and does not require generalisation. It is desirable to simplify the model while maintaining an "adequate representation" for π. There are two situations where the agent may merge the regions S' and S'':

- **Chain merge:** If $\text{succ}(S', \pi(S')) = S''$, $\text{succ}(S'', \pi(S'')) = S$, and $\pi(S') = \pi(S'')$ then merge S' with S''.
- **Sibling merge:** If $\text{succ}(S', \pi(S')) = S$, $\text{succ}(S'', \pi(S'')) = S$, and $\pi(S') = \pi(S'')$ then merge S' with S''.

For a deterministic model, $\text{succ}(S, a)$ denotes the region to which the agent transitions after taking action a from region S. These two kinds of merging, *chain merging* and *sibling merging*, are similar to the Squish algorithm[1] for simplifying goal-directed, reactive plans encoded as teleo-reactive trees (Nilsson, 1992). Figure 7.1 illustrates these two kinds of merging.

Chain and sibling merging may be generalised to nondeterministic environments. In nondeterministic environments, $\text{succ}(S, A)$ is not necessarily a single

[1]The Squish algorithm was developed by George H. John at Stanford University in 1994 but has not been published. The most detailed description of the algorithm is available in a paper by Nilsson (2000). The Squish algorithm was designed as a behavioural cloning technique, but the basic ideas are transferable to model simplification.

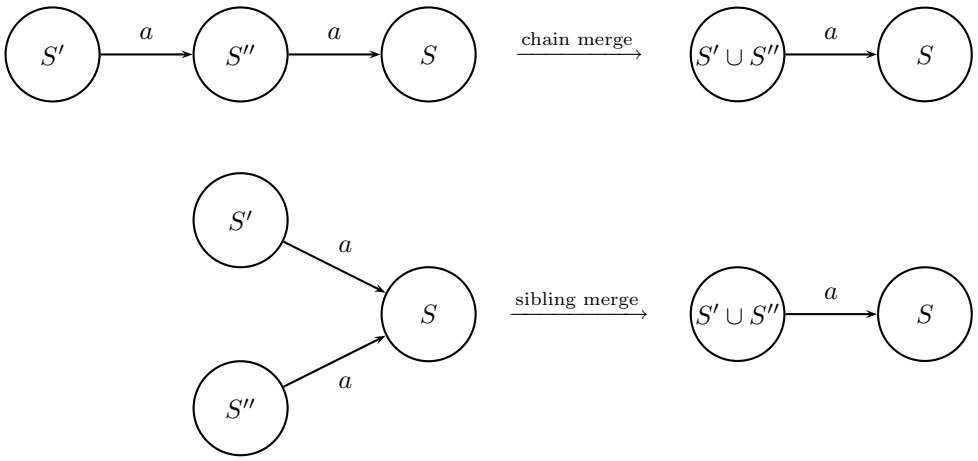


Figure 7.1: A demonstration of merging rules for state regions. The diagram shows the transformations of chain merging and sibling merging. The greedy actions are indicated along the transitions.

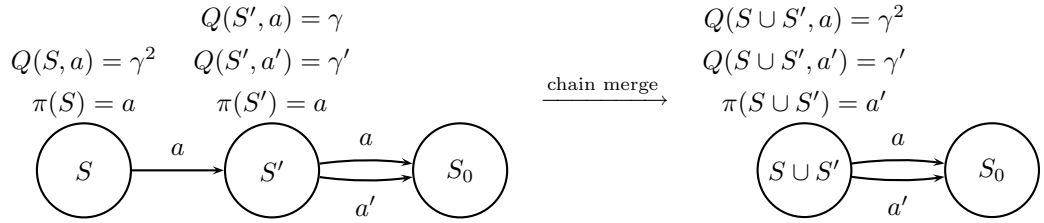


Figure 7.2: An example of how merging does not necessarily preserve the policy. The original model is estimated from two trajectories. The first trajectory involves $S \xrightarrow{a} S' \xrightarrow{a} S_0$, and the second trajectory involves $S' \xrightarrow{a'} S_0$. The policy changes after the merge if $\gamma^2 < \gamma' < \gamma$.

element, but the function may be adapted such that

$$\text{succ}(S, A) \equiv \begin{cases} S' & \text{if } P(S' \mid S, A) \approx 1 \\ \text{NIL} & \text{otherwise} \end{cases},$$

where ≈ 1 means within some small parameter ϵ.

Chain and sibling merging will not necessarily result in an equivalent plan. Since merging two regions changes the model, further planning on the new model can result in a different plan. Figure 7.2 provides a simple example of how the plan might change with chain merging. A similar example for sibling merging is not difficult to construct. Although merging can disrupt the policy, the disruption is not likely not hinder the performance of the agent in the long term so long as the agent is permitted to continue to revise its model.

In order to apply chain and sibling merging to problems with large action spaces, it is necessary to generalise the rules to:

- **Chain merge:** If $\text{succ}(S', \pi(S')) = S''$, $\text{succ}(S'', \pi(S'')) = S$, and $\pi(S') \sim \pi(S'')$, then merge S' with S''.
- **Sibling merge:** If $\text{succ}(S', \pi(S')) = S$, $\text{succ}(S'', \pi(S'')) = S$, and $\pi(S') \sim \pi(S'')$, then merge S' with S''.

In the earlier definition for small action spaces, the relation $A \sim A'$ denotes equality. However, it does not make sense to test for equality between two action regions associated with different state regions because different state regions partition the action space differently. The only way to compare A and A' is by consulting the experienced actions associated with these two action regions. AMPS tests to see if at least one of the examples associated with A can be mapped to A' and that one of the examples associated with A' can be mapped to A. Testing in this manner makes the relation $A \sim A'$ reflexive and symmetric and indicates that there is some overlap between the two regions.

When merging two state regions S and S', AMPS also merges the action regions associated with those two state regions. Although this means that the action regions must be repartitioned, the splitting process (Chapter 6) can quickly perform the necessary revision. An alternative is to simply use the partition associated with either S or S'.

AMPS supports the merging of action regions. Since it is not beneficial to make actional distinctions between suboptimal actions, AMPS merges non-greedy action regions. Although merging suboptimal action regions does no harm if the agent is certain of its policy, in the presence of uncertainty such a strategy can lead to longer convergence to an optimal policy. If the agent mistakes a suboptimal action region as an optimal region, AMPS might merge the actual optimal region with suboptimal regions. Discovering such a mistake may require substantial exploration.

7.3 Decision Graph Simplification

As the agent accumulates experience and revises the state and action space partitions, the decision graph supporting the partitions can become unnecessarily complex. AMPS incorporates several simplification algorithms to manage the

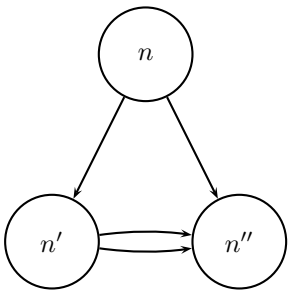


Figure 7.3: An example of how removing one redundant parent can lead to further redundant parents. In this illustration, the test associated with n' is redundant because both outcomes of the test lead to n''. When the simplification process replaces n' with n'', both outcomes of n will then lead to n'', causing another redundancy.

complexity of the decision graph. These simplification algorithms remove unnecessary internal nodes and adjust the links between nodes and their descendants in a way that does not alter the mapping of the sampled objects. This section motivates the simplification operations and explains their mechanism.

7.3.1 Removing Redundant Parents

The first simplification operation is based on a reduction procedure for binary decision diagrams (Bryant, 1986). If a node n has exactly one distinct child n', then the test associated with n is unnecessary. Therefore, n' may replace its redundant parent n without altering the mapping from objects to regions. This simplification procedure is extremely useful. When the agent has limited experience, it may try splitting a region only to later merge the resulting regions together again, causing redundancy in the decision graph.

If a decision graph does not contain redundant parents, the only way for this kind of redundancy to be introduced is with the merging of nodes. Hence, AMPS only needs to check for this redundancy when it merges nodes. AMPS merges nodes in two situations. The first situation is when the model needs to be simplified, as the previous section discusses. The second situation is when the decision graph itself is simplified. In fact, removing one redundant parent might reveal another redundant parent that also needs to be simplified (Figure 7.3).

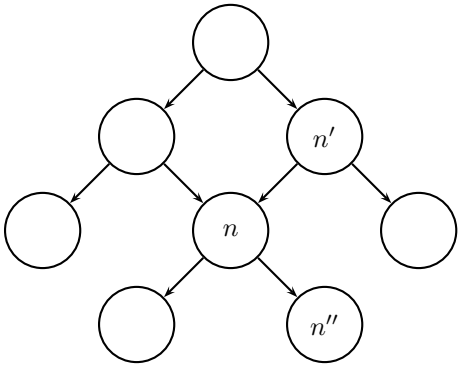


Figure 7.4: An example of a redundant split. If all of the objects that flow from n' to n end up associated with n'', then the link from n' to n should point directly to n''. Note that n may still point to n''.

7.3.2 Removing Redundant Splits

One kind of redundancy that appears frequently in practice involves a redundant test along the path from one node to another. Suppose that all the objects that flow through some branch at node n' end up in some node n'' after passing through node n (Figure 7.4). Instead of filtering objects through n, objects leaving the branch at node n' should go directly to n''. In order to identify and correct this redundancy, the agent must test for *exclusive flow* from a branch of n' through its child n to a grandchild n''. The function GET-EXCLUSIVE-FLOW-CHILD(n', n) returns a node n'', if one exists, to which all of the objects that pass from n' to n flow (Algorithm 5).

The REMOVE-REDUNDANT-SPLITS(n) procedure (Algorithm 6) uses the function GET-EXCLUSIVE-FLOW-CHILD to identify redundancies. Given some node n, typically the new internal node introduced by a split, the procedure creates a list L consisting of the parents of n. It then removes a node n' from L and checks to see if there is exclusive flow from n' through n and on to some child of n. If there is, then the branch leading from n' to n is redirected to point to the child with exclusive flow. The procedure continues until L is empty. If n no longer has any parents after this procedure, it is removed from the decision graph.

As shown in Section 10.3.3, this novel simplification algorithm is useful in reducing the complexity of the decision graph during incremental adaptation. This process is guaranteed to preserve the mapping of the sampled objects to leaf nodes, although the actual partition of the object space may change (Figure 7.5).

Algorithm 5 GET-EXCLUSIVE-FLOW-CHILD(n', n)

```
n* ← NIL
for all nodes n'' whose parent is n do
  for all objects x associated with n'' do
    if x flows through n' then
      if n* = NIL then
        n* ← n''
      else if n* ≠ n'' then
        return NIL
return n*
```

Algorithm 6 REMOVE-REDUNDANT-SPLITS(n)

```
create a list L consisting of the parents of node n
while L is not empty do
  remove any node n' from L
  n'' ← GET-EXCLUSIVE-FLOW-CHILD(n', n)
  if n'' ≠ NIL then
    change n' to lead to n'' instead of n
if n no longer has any parents then
  remove n from the graph
```

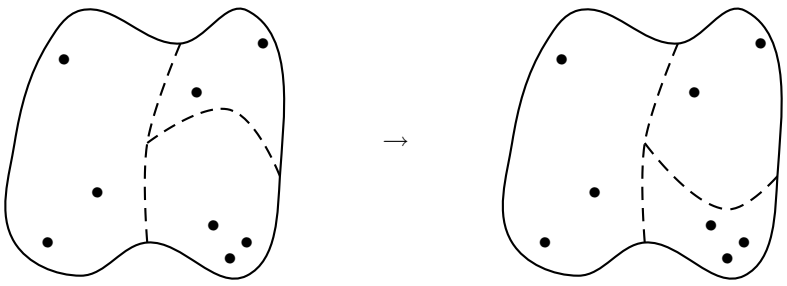


Figure 7.5: An example of the partition of the object space changing without affecting the mapping of sampled objects.

7.3.3 Reconstruction

Unfortunately, the algorithms earlier in this section are not sufficient for removing all redundancies from the decision graph. Figure 7.6 illustrates a decision graph with a redundancy. Some preliminary research has investigated ways to incrementally detect and remove other kinds of decision graph inefficiencies, but the methods developed are computationally expensive and therefore not suitable for most agents interacting with the world in real time.

Another route to take in simplifying decision graphs is rebuilding the decision graph from scratch based on the sampled objects stored in memory. The procedure first labels all of the stored instances with the region to which they belong. It then collapses the entire decision graph into a single root node. The algorithm then uses the split procedure from Section 6.4 to separate the samples associated with different labels. The procedure continues recursively on the resulting leaf nodes until each leaf contains samples with the same label. The procedure then merges together all leaf nodes containing samples with the same label.

Rebuilding the decision graph resembles standard decision tree induction with merging at the end. Of course, rebuilding the decision graph is quite expensive and should only be done when the agent has a large block of time available. In the experiments (Chapter 10), the system only rebuilds after each episode.

7.4 Discussion

This chapter presents two kinds of simplification. The first kind of simplification involves simplifying the model by merging state and action regions. The second kind of simplification involves simplifying the decision graph representation of the map. This section discusses the contributions of this chapter.

Model simplification is important for two reasons. First, a simpler model reduces the amount of computation required for planning. Second, a simpler model is easier to estimate from limited experience. Surprisingly, the other model-based reinforcement learning approaches in the literature (Section 4.2) do not incorporate model simplification; they only refine the discretisation of the state space. AMPS is the first model-based reinforcement learning algorithm that constructs its model through a dynamic process of splitting and merging existing regions of the state and action spaces.

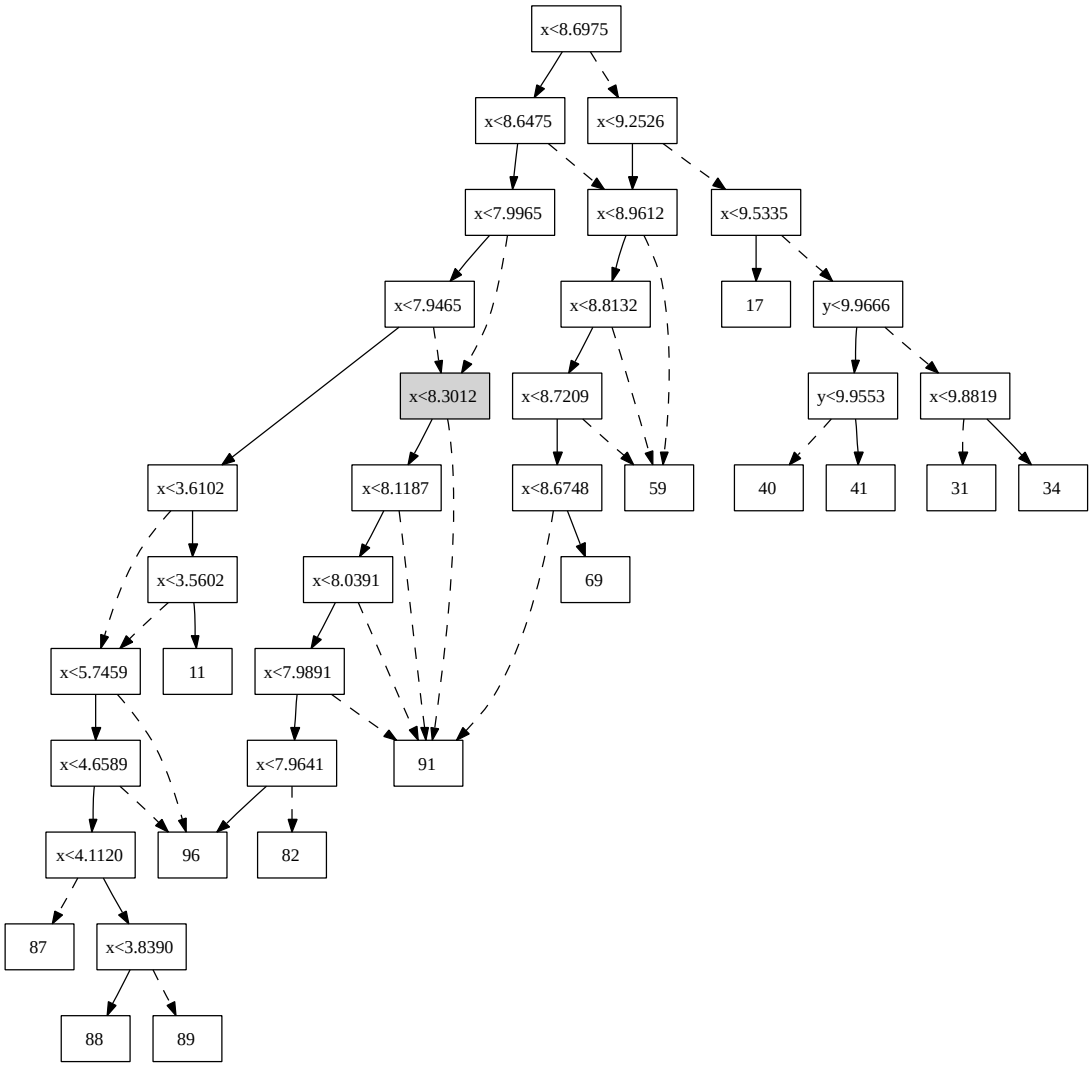


Figure 7.6: An example of a redundancy in a decision graph that was learned by AMPS. This decision graph partitions the Corner World state space. Notice the internal node labelled $x < 8.3012$ (shaded) that points to the internal node labelled $x < 8.1187$ and the leaf labelled 91. The internal node labelled $x < 8.1187$ and its child labelled $x < 8.0391$ are redundant. The same mapping may be represented without these two redundant nodes if $x < 8.3012$ is made to point directly to $x < 7.9891$ instead of $x < 8.1187$.

The model simplification algorithm (Section 7.2) in AMPS is a generalisation of the Squish supervised learning algorithm explored by George H. John and reported by Nilsson (2000) for constructing teleo-reactive trees. AMPS includes the first generalisation of Squish in a reinforcement learning context. AMPS extends the basic idea of Squish to nondeterministic transitions and partitioned action spaces.

It should be noted that model simplification in AMPS differs from the *aggregation* and *model minimisation* algorithms in the stochastic control literature that attempt to solve known MDPs by grouping together base states (e.g., Schweitzer et al., 1985; Bertsekas and Castañon, 1989; Givan et al., 2003). These methods require a full specification of the MDP at the base level, and are therefore unsuitable for model simplification in AMPS.

When using a decision graph to partition the state and action spaces, the splitting and merging processes can lead to unnecessarily complex decision graphs. It is important to simplify the decision graph as the agent accumulates experience and adjusts its model of the environment. A simpler decision graph is more computationally efficient, requires less memory, and is easier for a human to understand. AMPS includes algorithms for incremental decision graph simplification.

This chapter presents two online simplification procedures for decision graphs. The first procedure is based on a reduction operation for binary decision diagrams that removes redundant parents of a node (Bryant, 1986). The second procedure, which AMPS only performs after splitting a node, ensures that when there is exclusive flow of stored samples from one branch of a node to some other node, that branch leads directly to that node. This simplification algorithm is rather straightforward, but it is extremely useful in reducing the complexity of the decision graph.

The combination of the two online decision graph simplification algorithms works well in practice, but the decision graph will not necessarily be the simplest possible. Unfortunately, it is not likely that there exists an efficient algorithm for finding the simplest possible decision graph given labelled observations. Although efficient algorithms exist for minimising the number of nodes in an *ordered binary decision diagram*, finding the optimal ordering of tests is NP complete (Bollig and Wegener, 1996).

AMPS includes functionality for reconstructing decision graphs from scratch.

The procedure is similar to the standard tree induction algorithms with merging of identical leaf nodes at the end. Reconstruction is the only way to change the tests near the root of a decision graph. Unfortunately, this procedure is too expensive for AMPS to perform frequently since it requires processing all of the samples stored in memory. The resulting decision graph is not guaranteed to be optimal, which is not surprising because it is known that constructing an optimal decision tree is NP complete (Hyafil and Rivest, 1976). Although the decision graph may be more complex than necessary, the additional complexity does not significantly impact performance.

Chapters 5–7 have discussed how an agent can incrementally build an abstract model of the system dynamics from an incoming stream of experience. The next chapter explains how to efficiently plan over this model to produce competent behaviour.

Chapter 8

Planning

This chapter discusses planning over SMDPs with the assumption that the state and action spaces are finite and ignoring issues of generalisation. AMPS uses the techniques discussed in this chapter to generate reactive plans from learned models of the world. The first section introduces planning in general. The second section discusses value iteration methods for known models, and the third section presents adaptations and improvements to prioritised value iteration methods for estimated models. The fourth section describes experiments evaluating the performance of these methods on randomly generated SMDPs. The final sections present results and discuss findings.

8.1 Introduction

Planning is an important component of intelligence since it is the process by which an agent chooses a scheme of action to accomplish some objective. Planning requires a model, perhaps estimated from experience using techniques from Chapter 5. The model specifies how the world is expected to change in response to the behaviour of the agent.

Planning algorithms are typically designed to exploit a specific class of model. For example, Kushmerick et al. (1995) propose an approach that exploits the structure in a probabilistic STRIPS representation and Guestrin et al. (2003) suggest an approach that exploits the structure in a dynamic Bayesian network. This chapter assumes an SMDP representation.

A large portion of the artificial intelligence planning community is concerned with producing plans in the form of a sequence of actions from some designated

initial state. They typically assume a STRIPS-based formulation of the model represented with a description language like PDDL (McDermott, 2000; Fox and Long, 2003). Ghallab et al. (2004) provide a comprehensive survey of modern and classical planning techniques. A wide variety of approaches have been proposed, some involving planning graphs (Blum and Furst, 1997) or the reformulation of planning problems as propositional satisfiability problems (Kautz and Selman, 1992), constraint satisfaction problems (Do and Kambhampati, 2001), or heuristic search problems (Bonet and Geffner, 2001). Although these techniques have enjoyed successful application in a variety of domains, they are not suitable in general for an agent that is learning a model through online interaction with a highly stochastic world.

This thesis is concerned with planning techniques that result in reactive plans. Reactive plans, as mentioned earlier, are mappings from states to actions that an agent may use in a closed-loop, sense-act cycle. An efficient way to compute an optimal reactive plan given a model of the world is dynamic programming. As mentioned in Section 3.2, the bulk of the dynamic programming literature in the operations research and reinforcement learning communities focuses on MDPs. This thesis focuses on SMDPs, which are continuous-time generalisations of MDPs.

SMDPs have many practical applications including evaluating the performability of distributed processing systems (Ciardo et al., 1990), modelling communication in land mobile satellites (Bråten and Tjelta, 2002), evaluating power plant reliability (Perman et al., 1997), managing airline revenue (Gosavi et al., 2002), and solving factory optimisation problems (Mahadevan et al., 1997).

In the past ten years, the reinforcement learning community has become increasingly interested in SMDPs, partly because of their application to hierarchical reinforcement learning problems (surveyed by Barto and Mahadevan, 2003) where high-level behaviours require multiple time steps (Parr, 1998; Sutton et al., 1999; Dietterich, 2000) or take place in continuous time (Ghavamzadeh and Mahadevan, 2001). Bradtke and Duff (1995) generalised some of the standard reinforcement learning algorithms to SMDPs, including TD(λ) (Sutton, 1988), Q-learning (Watkins, 1989), and Adaptive Real-Time Dynamic Programming (Barto et al., 1995).

As the agent updates its estimates of the world model, it may use the dynamic programming technique called value iteration to find an optimal policy under the

assumption that the estimated model is correct (Bellman, 1957). Value iteration incrementally approximates the optimal value function, which can then be used to construct an optimal policy. Since the agent constantly updates its model during its interaction with the world, running value iteration to convergence at every step is not practical. Instead, updates of the value function must be prioritised. Moore and Atkeson (1993) originally developed an algorithm called prioritised sweeping that prioritises updates of the value function for MDPs, and Peng and Williams (1993) explored a similar algorithm called queue-Dyna. Wiering (1999) and Wingate and Seppi (2005) later made some enhancements to the prioritised sweeping algorithm. Recent work by McMahan and Gordon (2005) suggests a prioritisation approach for MDPs with positive costs that reduces to Dijkstra's algorithm (Dijkstra, 1959) when transitions are deterministic. These prioritised value iteration algorithms have proven to be extremely effective in practice on a variety of MDPs. Surprisingly, the generalisation of these algorithms to SMDPs is missing in the literature.

This chapter studies adaptive prioritised value iteration in SMDPs. Section 8.2 explains how to solve SMDPs when the world model is completely known, beginning with a discussion of value iteration and then introducing prioritised value iteration. Prioritised value iteration involves prioritising updates of the value function, thereby enhancing the speed of convergence to an optimal policy. The section presents three alternative algorithms, one based on that of Moore and Atkeson and two based on that of Wiering. Section 8.3 explains how these algorithms can be used to solve problems where the underlying world model is uncertain. When the model is not known a priori, the algorithms must adapt to a changing estimated world model. As proven in Appendix A, adaptive prioritised value iteration converges to the optimal value function when the agent uses suitable exploration and estimation strategies.

The second part of this chapter is an empirical study of adaptive prioritised value iteration in SMDPs. One of the major challenges of an empirical study is deciding which problems to use for evaluating performance. The first part of Section 8.4 discusses how to randomly generate problems and how their properties can be controlled by a small set of parameters. The remainder of the section discusses the various parameter settings for the algorithms and how to evaluate performance with a variety of metrics. Section 8.5 summarises the results of the experiments. The experiments compare the real-time convergence of various

algorithms and examine the effects of various estimation strategies on performance. Section 8.6 concludes with a discussion of the findings in this chapter. Appendix B explains the random generation of SMDPs in greater detail, and Appendix A contains the theorems and proofs referred to in this chapter.

8.2 Solution Methods

This section reviews the problem of solving SMDPs under the assumption that the agent possesses a complete and accurate model of the world dynamics and reward structure. Following a discussion of value iteration, this section presents three different prioritised value iteration algorithms.

8.2.1 Value Iteration

Section 3.2 defines B, known as the Bellman update operator, to be a mapping such that

$$BV(s) \equiv \max_{a \in \mathbb{A}(s)} \left[R(s,a) + \sum_{s' \in \mathbb{S}} P(s' \mid s,a)\gamma(s,a,s')V(s') \right]. \tag{8.1}$$

Theorem 2 states that V^* can be computed to any desired precision by repeatedly applying B to an arbitrary value function. Once V^* is known, an optimal policy π^* can be constructed as follows (see Theorem 3):

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

The value iteration algorithm (Algorithm 7) repeatedly applies B to an arbitrary value function until the L^∞-norm of the change in V, also known as the Bellman error magnitude, is less than Δ_{term}. If the resulting policy is to have an expected discounted return within ϵ of optimal from every state, then it is sufficient to set $\Delta_{\text{term}} = \epsilon(1-\alpha)/(2\alpha^2)$ as Theorem 4 states. Initially, V may be an arbitrary value function.

One may adapt the value iteration algorithm to increase the rate of convergence without having to keep both V and V' in memory. Algorithm 8 is the *Gauss-Seidel value iteration* algorithm (cf. Bertsekas and Tsitsiklis, 1997, pages 185–187), which iterates through the states according to some ordering and updates the value function based on the current value function. The stopping criterion is the same as Algorithm 7; if Δ_{term} is set to $\epsilon(1-\alpha)/(2\alpha^2)$ then

the expected discounted return of the greedy policy calculated from the value function at termination will be within ϵ of optimal (Theorem 5). Gauss-Seidel value iteration is a type of asynchronous value iteration. The next subsection considers asynchronous value iteration in greater depth where the updates are done in order of priority, where priority is related to how much the value function changed on previous updates.

Algorithm 7 Value Iteration

repeat
  $V' \leftarrow V$
  for all $s \in \mathbb{S}$ **do**
    $V(s) \leftarrow BV'(s)$
until $\|V' - V\| \leq \Delta_{\text{term}}$

Algorithm 8 Gauss-Seidel Value Iteration

repeat
  $\Delta_{\max} \leftarrow -\infty$
  for all $s \in \mathbb{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow BV(s)$
    if $|V(s) - v| > \Delta_{\max}$ **then**
      $\Delta_{\max} \leftarrow |V(s) - v|$
until $\Delta_{\max} \leq \Delta_{\text{term}}$

8.2.2 Prioritised Value Iteration

Prioritised value iteration is a type of *asynchronous value iteration* in which updates of the value function occur asynchronously, potentially on multiple processors (Bertsekas, 1982). So long as the value function is updated at each state infinitely often, the value function is guaranteed to converge to the optimal value function. A proof of convergence is a special case of that of Theorem 6 where the world model is known.

Prioritised value iteration can make value iteration much more efficient by prioritising the updates of the value function at individual states according to some heuristic. Moore and Atkeson (1993) suggest an approach called *prioritised*

sweeping and Wiering (1999) provides an adaptation of the algorithm. Both algorithms attempt to propagate updates of the value function through the state space. The prioritisation of these updates is related to how much the value function changes. Although both algorithms were originally designed for solving MDPs, they can be adapted to solve SMDPs.

The generic prioritised value iteration algorithm (Algorithm 9) starts by inserting all the states into the priority queue with priority zero in some random order. The algorithm then calls UPDATE on the highest priority states until the queue is empty. The UPDATE procedure is responsible for updating the value function and adapting the queue appropriately. The difference between the approach of Moore and Atkeson and that of Wiering is in the UPDATE procedure.

Algorithm 9 Prioritised Value Iteration

$H \leftarrow \emptyset$
for all $s \in \mathbb{S}$ **do**
  $\Delta(s) \leftarrow 0$
  INSERT$(H, s, \Delta(s))$
while $H \neq \emptyset$ **do**
  $s \leftarrow$ EXTRACT-MAX(H)
  $\Delta(s) \leftarrow 0$
  UPDATE(H, Δ, s)

The priority queue supports the following operations:

- INSERT(H, s, p) inserts state s into the queue H with priority p.
- EXTRACT-MAX(H) extracts the highest priority state from the queue H.
- CONTAINS(H, s) indicates whether state s is contained in the queue H.
- INCREASE-PRIORITY(H, s, p) increases the priority of state s to p in the queue H.
- REMOVE(H, s) removes state s from the queue H.

Since the size of the state space is known, CONTAINS may be computed in constant time using a binary array. The INSERT and INCREASE-PRIORITY operations require only constant time with a relaxed heap (Driscoll et al., 1988) or

amortised constant time with a Fibonacci heap (Fredman and Tarjan, 1987).[1] EXTRACT-MAX and REMOVE run in $O(\log n)$ time with a relaxed heap and $O(\log n)$ amortised time on a Fibonacci heap.

The algorithm proposed by Moore and Atkeson for prioritised value iteration computes an upper bound on the amount the value function at each state will change due to the change in value function of one of its successors. The upper bound at state s is denoted $\Delta(s)$ and serves as the priority of state s in a priority queue. Each iteration of the algorithm extracts the highest priority state from the priority queue, updates its value, and updates the priorities of its predecessors (i.e. the states that lead immediately to that state). In order to perform the updates quickly, the algorithm maintains the predecessor set for each state. The predecessor set for a state s is given by $\text{pred}(s)$ and contains a set of tuples containing every state and action pair leading to s.

Algorithm 10 is based on the one given by Moore and Atkeson, adapted for solving SMDPs when the model is known a priori. Using an efficient priority queue implementation, this algorithm runs in constant time, assuming that the number of predecessors and successors for all states is less than a small constant, which is typically the case in interesting problems. The constant ϵ controls the update precision and is a way to balance speed with accuracy. To find exact solutions, as is done in the experiments discussed in Section 8.5, ϵ can be set to zero.

Wiering's algorithm (Algorithm 11) is designed as an improvement to the algorithm proposed by Moore and Atkeson. The semantics of Δ are slightly different. In Wiering's algorithm, $\Delta(s)$ is a measure of how much $V(s)$ changed since updating its predecessors. Because $V(s)$ can both increase and decrease before the predecessors are updated, it is possible for $\Delta(s)$ to be positive or negative. The priority of updating the predecessor of a state is equal to the absolute value of $\Delta(s)$. To make updates fast, a predecessor set is maintained for each state. The predecessor set for a state s is given by $\text{pred}(s)$ and contains a set of states leading immediately to s.

With an efficient heap implementation, Wiering's update procedure requires $O(\log n)$ time. Although each update of Algorithm 11 is asymptotically more

[1] Moore and Atkeson and Wiering suggest using a heap data structure, which supports INSERT and INCREASE-PRIORITY in $O(\log n)$ time (Cormen et al., 2001, pages 138–141). Although the heap data structure is much easier to implement than a relaxed heap or a Fibonacci heap, it does not provide the same asymptotic performance.

Algorithm 10 UPDATE(H, Δ, s) (cf. Moore and Atkeson, 1993)

```
v ← V(s)
V(s) ← max_{a∈𝔸(s)} [R(s,a) + Σ_{s'∈𝕊} P(s' | s,a)γ(s,a,s')V(s')]
for all (s', a') ∈ pred(s) do
  p ← P(s | s', a')γ(s', a', s) |V(s) − v|
  if p > Δ(s') then
    Δ(s') ← p
    if Δ(s') > ε then
      if CONTAINS(H, s') then
        INCREASE-PRIORITY(H, s', Δ(s'))
      else
        INSERT(H, s', Δ(s'))
```

expensive than Algorithm 10, Algorithm 11 usually updates the value function at more than just one state and can therefore lead to convergence with fewer updates.

This thesis proposes Algorithm 12 as an improvement to Algorithm 11. The changes are rather subtle but decrease the cost of an update from logarithmic to constant. Whenever the change in value function at a state requires the priority to be decreased in Algorithm 11, Algorithm 12 simply ignores the update to Δ and the priority. This faster version only increases priorities and inserts items, allowing the algorithm to run in constant time with a Fibonacci heap or relaxed heap. Of course, $\Delta(s)$ no longer represents the exact amount $V(s)$ has changed since UPDATE was called on s, but the number of iterations until convergence of this algorithm does not vary significantly with Algorithm 11 and the real-time performance of this algorithm is significantly better as shown in Section 8.5.

8.3 Adaptive Prioritised Value Iteration

The prioritised value iteration algorithms discussed in the previous section can be adapted to problems where the agent has no prior knowledge of the world model. As Section 2.6.2 mentions, there are two different classes of methods for solving problems when the underlying model is unknown. The Bayesian approach involves specifying a prior distribution over possible models and revising this distribution in accordance with information gained through experience with the

Algorithm 11 UPDATE(H, Δ, s) (cf. Wiering, 1999)

for all $s' \in \text{pred}(s)$ **do**
  $v \leftarrow V(s')$
  $V(s') \leftarrow \max_{a' \in \mathbb{A}(s')} \left[R(s', a') + \sum_{s'' \in \mathbb{S}} P(s'' \mid s', a') \gamma(s', a', s'') V(s'') \right]$
  $p \leftarrow \Delta(s')$
  $\Delta(s') \leftarrow \Delta(s') + V(s') - v$
  if $|\Delta(s')| > \epsilon$ **then**
    if CONTAINS(H, s') **then**
      if $|\Delta(s')| > |p|$ **then**
        INCREASE-PRIORITY$(H, s', |\Delta(s')|)$
      else
        REMOVE(H, s')
        INSERT$(H, s', |\Delta(s')|)$
    else
      INSERT$(H, s', |\Delta(s')|)$
  else
    if CONTAINS(H, s') **then**
      REMOVE(H, s')

Algorithm 12 UPDATE(H, Δ, s)

for all $s' \in \text{pred}(s)$ **do**
  $v \leftarrow V(s')$
  $V(s') \leftarrow \max_{a' \in \mathbb{A}(s')} \left[R(s', a') + \sum_{s'' \in \mathbb{S}} P(s'' \mid s', a') \gamma(s', a', s'') V(s'') \right]$
  $p \leftarrow \Delta(s')$
  $\Delta(s') \leftarrow \Delta(s') + V(s') - v$
  if $|\Delta(s')| > \epsilon$ **then**
    if CONTAINS(H, s') **then**
      if $|\Delta(s')| > |p|$ **then**
        INCREASE-PRIORITY$(H, s', |\Delta(s')|)$
      else
        $\Delta(s') \leftarrow p$
    else
      INSERT$(H, s', |\Delta(s')|)$
  else
    $\Delta(s') \leftarrow p$

world. Actions are chosen so as to maximise expected discounted reward over the possible world models. Non-Bayesian methods attempt to arrive at optimal policies in the limit as experience is accumulated.[2]

Because the Bayesian approach is not practical for large problems, this thesis focuses on a non-Bayesian approach where the current estimated model is assumed to be the true model. Prioritised value iteration can provide an estimate of the optimal policy from an estimated model. Although it seems reasonable to simply follow the current estimated optimal policy, doing so will not guarantee convergence to the true optimal policy (for a counterexample see Kumar and Becker, 1982, Example 12). In general, it is necessary to adopt an exploration strategy that allows the estimated model to converge to the true model. For the model estimation methods in Section 5.3, any exploration policy that allows any valid action to be taken at any state with non-zero probability is sufficient for convergence.

Full value iteration is not necessary every time the estimated model changes. Adaptive real-time dynamic programming (Barto et al., 1995) only updates the value function at the most recently visited state. The prioritised sweeping algorithm (Moore and Atkeson, 1993) for MDPs performs prioritised updates in a manner similar to that of Algorithm 10 when there is time available between decisions. Wiering (1999) uses a similar process for MDPs.

Adaptive prioritised value iteration (Algorithm 13) incrementally improves its estimate of the world model and the optimal value function while interacting with the world. After executing an action according to some exploration strategy (Section 5.5.1) and observing the transition to the next state, it updates the estimate of the world model (Section 5.3) and promotes the most recently visited state to the front of the queue. In between transitions, the algorithm calls UPDATE on the highest priority states until either the queue is empty or some other condition is met. It is common to limit the number of updates per transition to some small constant so that real-time response can be maintained.

Provided that the agent employs an appropriate exploration strategy and its estimate of the world model strongly converges to the true model, adaptive prioritised value iteration will strongly converge to the optimal value function (Theorem 6 in Appendix A).

[2]Barto et al. (1995, Section 7) discuss the difference between Bayesian and non-Bayesian approaches to adaptive control and Kumar (1985) provides a comprehensive survey.

Algorithm 13 Adaptive Prioritised Value Iteration

loop
  Execute an action according to some exploration strategy
  Update the model estimate according to the state transition and reward
  $s \leftarrow$ most recent state
  if CONTAINS(H, s) **then**
    INCREASE-PRIORITY(H, s, ∞)
  else
    INSERT(H, s, ∞)
  while $H \neq \emptyset$ **do**
    $s \leftarrow$ EXTRACT-MAX(H)
    $\Delta(s) \leftarrow 0$
    UPDATE(H, Δ, s)

8.4 Experiments

Although all the algorithms in the previous section will find the optimal policy in the limit under some rather weak assumptions, it is important to evaluate their performance empirically. This section discusses how random problems can be generated and used to test both the non-adaptive and adaptive algorithms. Included in this section is an explanation of the experimental parameters and performance metrics.

8.4.1 Random Problem Generation

One of the most significant challenges of evaluating an algorithm is choosing the test problems. The work by Moore and Atkeson (1993) and the work by Wiering (1999) use grid worlds fashioned as MDPs. Moore and Atkeson also investigate randomly generated stochastic problems on the unit square, which serves as inspiration for the experimental problems of this chapter.

In order to define an SMDP, it is necessary to specify $P(s' \mid s, a)$, $P_t(t \mid s, a, s')$, $P_r(r \mid s, a, s')$, and $P_\rho(\rho \mid s, a, s')$. Most interesting problems with real-world applications are not fully connected, or in other words, $P(s' \mid s, a) = 0$ for most values of s' given s and a.[3] Moore and Atkeson choose some random number of successors and then assign random transition probabilities to these successors.

[3] A notable exception is the SysAdmin domain described by Guestrin et al. (2003).

To generate a random problem, one can decide upon parameterised probability distributions for $P_t(t \mid s, a, s')$, $P_r(r \mid s, a, s')$, and $P_\rho(\rho \mid s, a, s')$ such as the normal or exponential distributions and assign randomly chosen parameters.

The experiments use a random generator to produce challenging and realistic problems. The complete specification of the generator is in Appendix B. Generating an SMDP involves randomly choosing points in a d-dimensional hypercube to serve as states. Each state connects to its k nearest neighbours that are not obstructed by randomly-placed hyperspheres. Actions correspond to the cardinal directions, and the probability of transitioning from a state to one of its neighbours by an action is proportional to the cosine of the angle formed between the states and the cardinal direction associated with the action. The expected duration required to transition from one state to another is related to the Euclidean distance between the states. The generator takes eight parameters that affect the types of problems it generates.

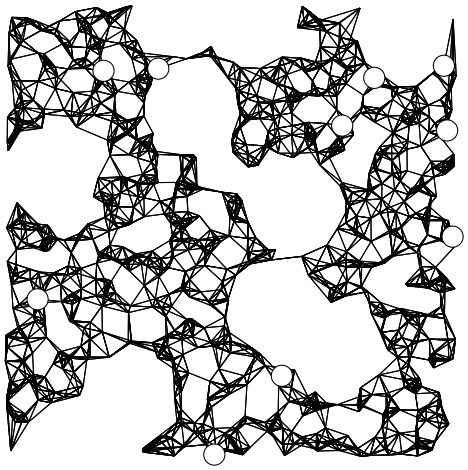
- d, the number of dimensions. Figure 8.4.1 illustrates randomly generated SMDPs in two and three dimensions. The number of actions available to the agent is $2d$, corresponding to the cardinal directions in d-dimensional space. The experiments involve problems in two or three dimensions since they are the easiest to visualise.

- n_{state}, the number of states. The experiments involve 100 to 10,000 states. A simple way to increase the difficulty of a problem is to increase the number of states.

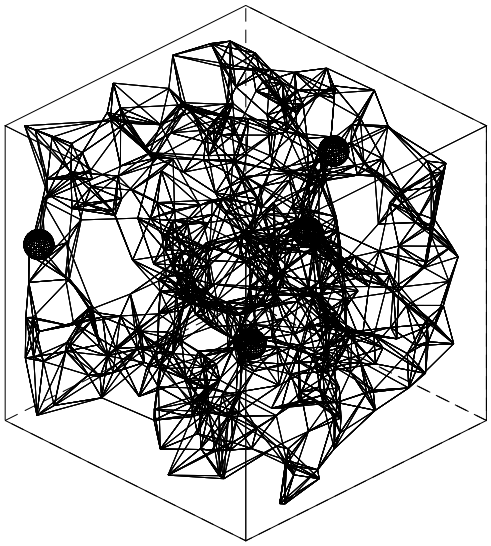
- n_{obstacle}, the number of hyperspheres that serve as obstacles. All experiments in this chapter have ten obstacles. Obstacles make for interesting problems. As Figure 8.4.1 illustrates, especially in the two-dimensional problem, there are narrow "bridges" connecting regions of the state space between obstacles.

- r, the radius of the hyperspheres. In the experiments, the radii are set to 0.1 units. Larger values of r are more likely to disconnect the state space.

- n_{reward}, the number of terminal reward states (known as goal states). The agent receives a unit lump-sum reward when transitioning to one of these states. The experiments involve either one or ten goal states. Typically, the more goal states there are, the easier the problem.



(a) A randomly generated SMDP in 2D consisting of 1000 states. The terminal states are indicated by circles.



(b) A randomly generated SMDP in 3D consisting of 500 states. The terminal states are indicated by spheres.

Figure 8.1: Randomly generated SMDPs. The vertices represent states and edges between vertices indicate that for some action there is a non-zero probability of transitioning between states. Both SMDPs are absorbing.

- k, the number of neighbours to consider as successors. All experiments consider eight neighbours.

- ϵ, the standard deviation of lump-sum reward when reaching a goal state and the standard deviation of reward rates. The agent receives zero lump-sum reward except when transitioning to a goal state. All transitions acquire reward at a rate selected randomly from a normal distribution with mean -1 and standard deviation ϵ. This parameter only affects the difficulty of problems when the underlying world model is unknown. All experiments use 0.1 for ϵ.

Some parameter settings make it unlikely that the first problem generated is *absorbing*. Absorbing SMDPs are problems where a terminal state is eventually reachable from any state in the state space with non-zero probability. The generator randomly creates problems until it finds one that is absorbing.

8.4.2 Algorithms

The experiments compare the performance of both non-adaptive and adaptive algorithms. The non-adaptive algorithms include:

- **VI**: Value iteration, Algorithm 7.

- **GS**: Gauss-Seidel value iteration, Algorithm 8

- **MA**: Prioritised value iteration based on that of Moore and Atkeson, Algorithms 9 and 10.

- **W**: Prioritised value iteration based on that of Wiering, Algorithms 9 and 11.

- **FW**: A faster version of W, Algorithms 9 and 12.

The adaptive algorithms include:

- **ART**: Adaptive real-time dynamic programming (Barto et al., 1995), where the agent only updates the value of the most recently visited state.

- **AMA**: Adaptive prioritised value iteration based on that of Moore and Atkeson, Algorithms 13 and 10.

- **AW**: Adaptive prioritised value iteration based on that of Wiering, Algorithms 13 and 11.
- **AFW**: A faster version of AW, Algorithms 13 and 12.
- **AGS**: Gauss-Seidel value iteration over the estimated model, Algorithm 8.

The algorithms requiring a priority queue use a Fibonacci heap as the supporting data structure.

8.4.3 Exploration Strategies

The experiments use an ϵ-greedy exploration strategy (Section 5.5.1) with $Q^{\max}$ from Equation 5.1. The default value for ϵ is 0.1, and the default value for n_{bored} is 5. The experiments employ this exploration policy because of its simplicity and because it encourages an initial high-rate of random exploration. The experiments do not involve any special tuning of these parameters for the various problems.

8.4.4 Performance Evaluation

One may use a variety of metrics to measure the quality of an estimated value function V or estimated greedy policy π relative to the optimal value function V^* or optimal decision rule π^*. Some of the most natural metrics use the L^1 or L^2 norms. For example, $\|V - V^*\|_1$ gives the sum of the differences between V and V^*, and $\|V - V^*\|_2^2$ gives the sum of the squared differences between V and V^*.

Instead of comparing V and V^*, it might be more useful to compare V^π with V^*. This comparison is helpful in determining how effective following the greedy policy of the estimated value function is compared to following the optimal policy. It is often the case that V^π converges to V^* long before V converges to V^*. Unfortunately, computing V^π is about as difficult as computing V^*.

A fast way to determine the quality of π is by measuring the Hamming distance between π and π^*. Here, $H(\pi, \pi^*)$ denotes the fraction of states at which the two decision rules agree. Moore and Atkeson (1993) and Wiering (1999, Section 4.4.3) use this metric. However, although $H(\pi, \pi^*)$ might not be close to 1, $\|V^\pi - V^*\|_2^2$ might be close to 0. In other words, even though π and π^* disagree in many states, it might still be the case that π is expected to achieve near-optimal reward.

To compare the performance of non-adaptive algorithms, one may evaluate these metrics over a series of iterations. One iteration of VI or GS involves a full

sweep of the state space, one iteration of M involves updating a single state, and one iteration of W or FW involves updating potentially a few states. The amount of computation required per iteration varies greatly between algorithms. Hence, it is important to also evaluate these metrics over real time. However, real-time evaluation is not without its problems since the performance of the algorithm can be sensitive to its implementation and the specification of the system for evaluation.

This chapter evaluates the performance of adaptive algorithms over real time and simulated time. Simulated time refers to the time spent transitioning from state to state. Simulated time is not, of course, sensitive to the details of the implementation or the specification of the system used for evaluation. This chapter also measures performance relative to the number of decisions made while interacting with the world.

8.5 Results

This section summarises some of the results of various experiments. All algorithms are implemented in Java 5 and a reasonable effort was made to ensure efficiency. All real-valued numbers are represented using a double precision floating point representation occupying 8 bytes. The experiments were conducted on a 2.4 GHz Pentium 4 workstation with 512 MB of RAM. All experiments use a 0.01 continuous compound discount rate and $n_{\text{obstacle}} = 10$, $r = 0.1$, $k = 8$, and $\epsilon = 0.1$.

8.5.1 Non-Adaptive Algorithms

One set of experiments tests the non-adaptive algorithms (i.e. VI, GS, MA, W, and FW) on randomly generated two-dimensional problems with a single goal state as discussed in Section 8.4.1. Figures 8.2 and 8.3 show the convergence curves for the non-adaptive algorithms on a sample problem with 500 states. Table 8.1 summarises the results of experiments on five different problems with varying numbers of states.

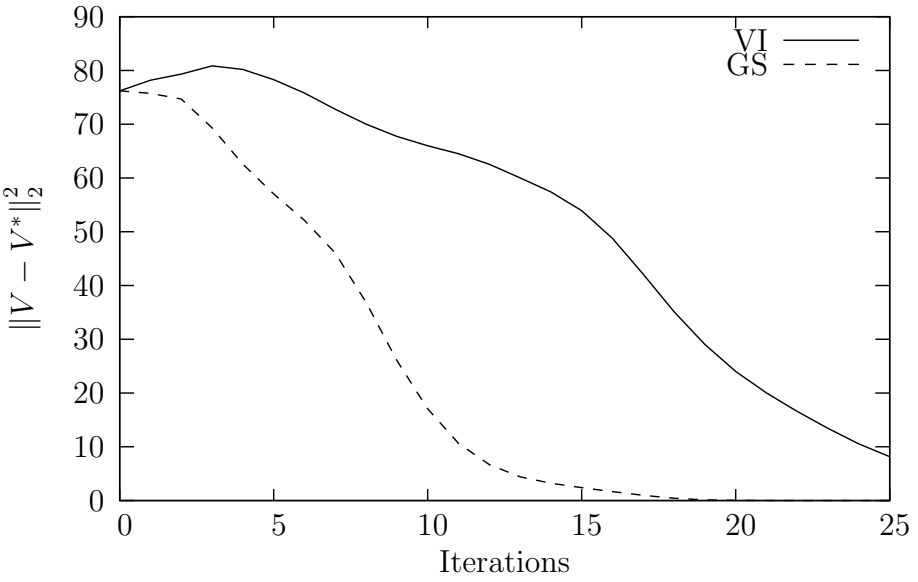
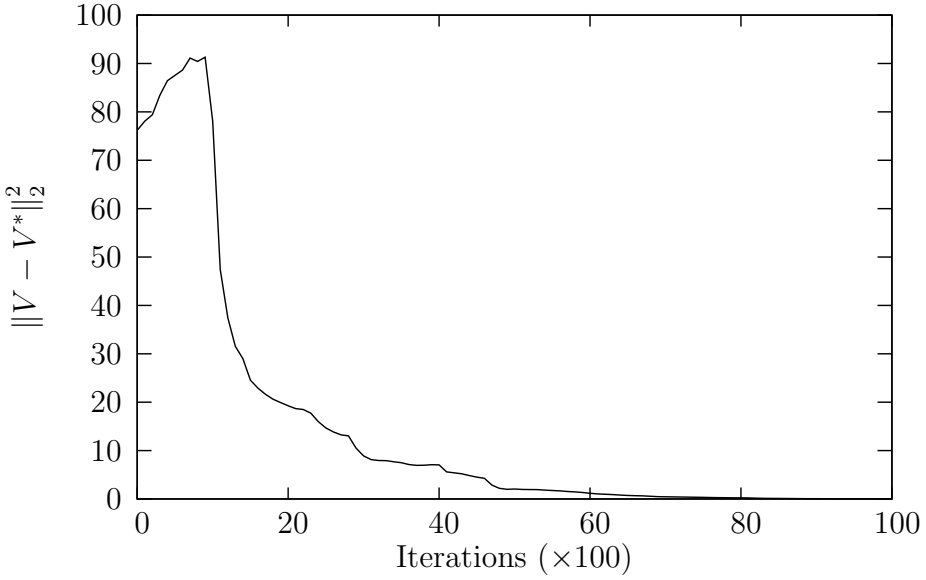


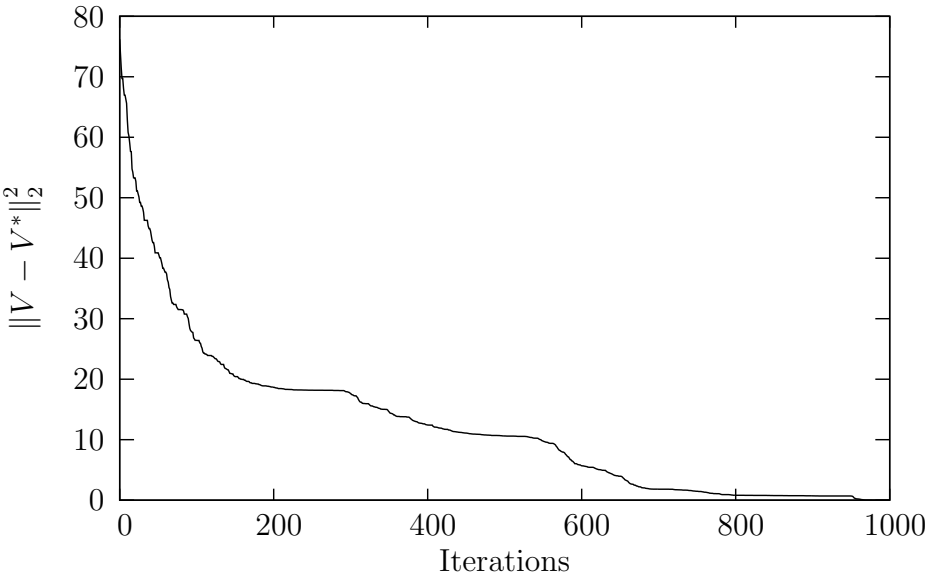
Figure 8.2: Convergence curves for the value iteration algorithms, VI and GS. The vertical axis indicates how close the computed value function is to the optimal value function. An iteration consists of a full sweep.

states	VI	GS	MA	W	FW
100	0.85±0.04	0.50±0.00	4.26± 0.32	3.32±0.20	1.55± 0.10
500	12.86±0.02	6.67±0.01	66.67± 0.11	25.71±0.06	11.00± 0.02
1000	51.88±0.07	28.42±0.08	258.49± 1.74	145.85±0.20	66.33± 0.10
5000	522.18±0.50	258.68±0.18	3823.31±14.54	1444.34±1.43	718.81± 0.52
10000	1615.41±8.55	785.41±0.36	12656.40±16.55	3792.05±8.79	2048.87±12.17

Table 8.1: Expected runtime in milliseconds for the non-adaptive algorithms to find the optimal policy estimated from 100 sample runs. Indicated are 99% confidence intervals.



(a) MA curve.



(b) W curve.

Figure 8.3: Convergence curves for the prioritised value iteration algorithms, MA and W. The vertical axis indicates how close the computed value function is to the optimal value function. An iteration consists of processing a single state that has been removed from the front of the priority queue. The curve for FW is indistinguishable from W on this scale.

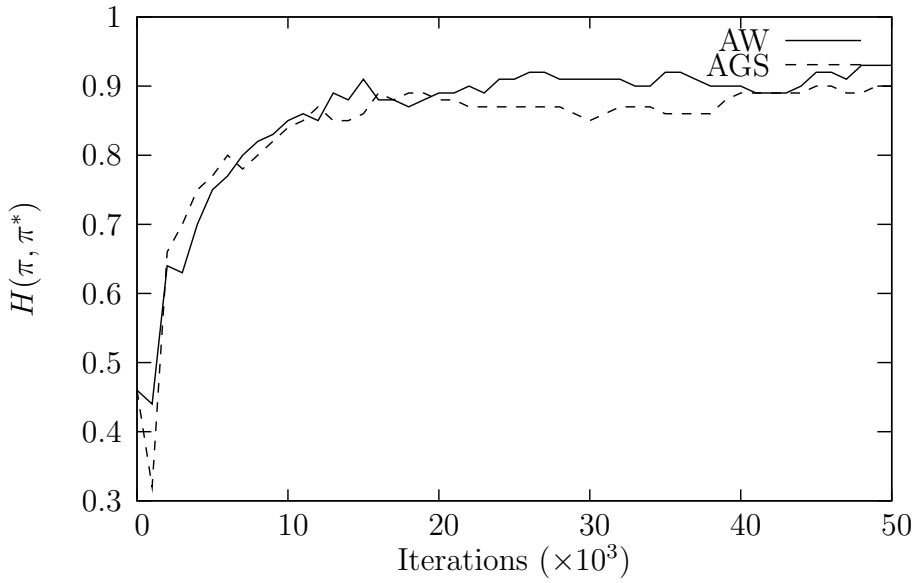


Figure 8.4: Performance curve for AW against AGS, both using parametric estimation. These curves show no significant difference between the two algorithms. Adaptive algorithms learn near-optimal policies quickly, but going from 90% to 100% optimal often requires a tremendous amount of exploration.

8.5.2 Estimation

Experiments were conducted to test the performance of the various adaptive algorithms with parametric and non-parametric estimation. These experiments involve problems with 1000 states in two dimensions and 10 goals.

The first experiment tests how many goals the agent can find within 1 million seconds of simulated time using different adaptive prioritised value iteration algorithms and estimation strategies. Table 8.2 summarises the results.

Table 8.2 reveals that there is no significant difference between the performance of non-parametric estimation and parametric estimation. There is also no apparent difference in performance between the adaptive prioritised planning algorithms. Performing full Gauss-Seidel value iteration to convergence is extremely expensive but provides no benefit over the adaptive prioritised value iteration algorithms (see Figure 8.4 for sample performance curves). Although ART achieves almost as many goals as the adaptive prioritised value iteration algorithms, the convergence of its estimate of the value function is slower. In the limit, however, ART is indistinguishable from AMA, AW, and AFW.

	goals		$\|V - V^*\|_2^2$		$H(\pi, \pi^*)$		time (s)	
	P	NP	P	NP	P	NP	P	NP
10^2 s of simulation time (optimal with exploration: 177.86 ± 1.00 goals, random: 26.36 ± 0.50 goals)								
ART	26.88± 0.50	27.05± 0.53	21.57±0.45	21.63±0.45	0.33±0.00	0.33±0.00	0.00±0.00	0.00±0.00
AMA	26.78± 0.52	26.82± 0.52	15.17±0.33	15.13±0.37	0.33±0.00	0.33±0.00	0.01±0.00	0.01±0.00
AW	27.19± 0.53	27.05± 0.52	13.66±0.29	13.74±0.32	0.33±0.00	0.33±0.00	0.01±0.00	0.01±0.00
AFW	27.11± 0.51	26.71± 0.49	13.61±0.29	13.56±0.35	0.33±0.00	0.33±0.00	0.01±0.00	0.01±0.00
AGS	26.98± 0.50	26.86± 0.51	13.59±0.45	13.58±0.28	0.33±0.00	0.33±0.00	0.04±0.00	0.04±0.00
10^3 s of simulation time (optimal with exploration: 1782.16 ± 3.18 goals, random: 277.78 ± 1.62 goals)								
ART	716.61± 3.24	714.01± 3.17	1.16±0.03	1.18±0.03	0.62±0.00	0.62±0.00	0.02±0.00	0.02±0.00
AMA	723.20± 3.01	722.78± 3.17	1.04±0.03	1.05±0.03	0.62±0.00	0.62±0.00	0.05±0.00	0.05±0.00
AW	722.76± 3.10	721.80± 3.10	1.01±0.03	1.02±0.03	0.62±0.00	0.62±0.00	0.08±0.00	0.08±0.00
AFW	722.16± 3.19	722.43± 3.18	1.01±0.03	1.02±0.03	0.62±0.00	0.62±0.00	0.06±0.00	0.07±0.00
AGS	723.54± 3.06	721.93± 3.13	0.99±0.03	1.04±0.03	0.62±0.00	0.62±0.00	1.02±0.00	1.02±0.00
10^4 s of simulation time (optimal with exploration: 17933.72 ± 10.43 goals, random: 2766.58 ± 5.25 goals)								
ART	12796.65±22.14	12790.68±23.38	0.09±0.00	0.09±0.00	0.79±0.00	0.79±0.00	0.17±0.00	0.18±0.00
AMA	12806.79±24.46	12794.84±23.89	0.10±0.00	0.10±0.00	0.78±0.00	0.78±0.00	0.33±0.00	0.34±0.00
AW	12804.56±24.28	12806.56±24.03	0.09±0.00	0.10±0.01	0.78±0.00	0.79±0.00	0.59±0.00	0.60±0.00
AFW	12803.01±22.98	12800.15±24.63	0.09±0.00	0.09±0.00	0.78±0.00	0.78±0.00	0.54±0.00	0.56±0.00
AGS	12808.65±24.62	12812.43±22.81	0.09±0.00	0.09±0.00	0.78±0.00	0.78±0.00	15.01±0.03	15.02±0.02

Table 8.2: Expected performance according to various metrics estimated from 1000 runs. Results are shown after 10^2 s, 10^3 s, and 10^4 s of simulation time. The agent makes approximately 5 decisions per second. Indicated are 99% confidence intervals.

8.5.3 Trajectory Interruption

Several experiments involve measuring the convergence of the estimated policy to the optimal policy when trajectories are interrupted after 0.1 s of simulation time, resulting in the interruption of 63% of the transitions. These experiments use a three-dimensional world with 100 states and a single goal.

The experiments reveal that the parametric method that takes into account censored data as described in Section 5.3.3 is required for satisfactory results. If censored duration measurements are ignored, regular parametric and non-parametric methods are able to find 90% optimal policies but not nearly as quickly as the adapted parametric method. When using FW, for example, the average amount of simulation time required to find a 90% optimal policy over 100 runs is 978 s and 1124 s for regular parametric and non-parametric methods respectively but only 349 s for the parametric method that takes into account censoring. The regular parametric and non-parametric methods do not find 95% optimal policies, but the adapted parametric method finds one in 5055 s of simulation time.

8.6 Discussion

The purpose of this chapter is to explore prioritised value iteration algorithms over SMDPs for use as the plan-revision process in AMPS. Prioritised value iteration is important when the estimate of the underlying dynamics changes frequently and the computational power of the agent cannot cope with replanning over the entire state space. The agent needs to be able to quickly identify which regions of the state space require replanning. This chapter focuses on planning in isolation and ignores issues with generalisation by assuming discrete state and action spaces.

This chapter generalises the prioritised value iteration algorithms of Moore and Atkeson (1993) and Wiering (1999) and suggests a way to further improve these algorithms. These algorithms were originally designed to solve discrete-time MDPs, and this chapter adapts them for use with continuous-time problems. The adaptation involves evaluating integrals over the distribution over the transition durations. Although other researchers (e.g., Bradtke and Duff, 1995) have generalised existing MDP algorithms to SMDPs, this thesis is the first to generalise the work by Moore and Atkeson (1993) and Wiering (1999) to SMDPs. Convergence proofs are in the appendix.

The first part of this chapter evaluates the planning algorithms on known models. On the randomly generated problems, Gauss-Seidel value iteration converges to the optimal value function faster in real time than the algorithms relying on priority queues. Although the prioritised value iteration algorithms require a fraction of the updates of the value function required by Gauss-Seidel value iteration, each removal of the highest priority element from the queue is costly.[4] Real-time comparisons of the prioritised value iteration algorithms reveal that W outperforms MA on large problems, and FW is about twice as fast as W.

The main focus of this chapter, however, is on planning over an estimated model that changes over time, not a known model. As Section 5.3 discusses, one may use a parametric or non-parametric approach to estimate an SMDP model from experience. One might expect that parametric estimation would result in a better estimate of the underlying model because it utilises prior knowledge about the form of the distribution. However, as the experiments in this chapter reveal, there is no significant difference in the performance of parametric estimation and non-parametric estimation—so long as all trajectories are allowed to complete.

If trajectories are interrupted, it is important to use a parametric estimation strategy that takes into account censoring (Section 5.3.3). Without taking into account censoring, the agent may not be able to estimate a model that is accurate enough for it to extract an optimal policy. Although parametric estimation with censored data is frequently done in reliability analysis and other areas, this thesis appears to be the first to exploit censored data in a reinforcement learning context.

The experiments in this chapter reveal that the adaptive prioritised value iteration algorithms perform marginally better than the real-time dynamic programming algorithm. There is no significant difference in the performance of the adaptive prioritised value iteration algorithms, although it should be noted that it takes less time to execute an iteration of AMA than W or FW. Interestingly, performing full Gauss-Seidel value iteration with each update of the estimated model does not improve performance. Planning until convergence at each step is not useful and is very costly, especially in problems with large state spaces.

In AMPS, the prioritised value iteration algorithm performs planning over state and action regions. The modelling process is likely to split and merge these regions many times during the lifetime of the agent. A single split or merge can

[4] Wingate and Seppi (2005) make a similar observation and suggest various ways of improving performance including variable reordering and partitioning. Their work focuses exclusively on problems with a known model.

drastically affect the structure of the estimated model, and the planning process must be able to quickly revise the plan. Prioritising the updates of the value function allows the agent to make the most important plan revisions first. The next chapter discusses how the prioritised planning process integrates with the modelling process in AMPS.

Chapter 9

Integration

This chapter explains how to integrate the modelling and planning ideas presented in the earlier chapters to produce an agent that learns to behave competently in dynamic environments. The first section introduces the general reinforcement learning framework that supports the implementation of AMPS. The second and third sections discuss the data structures and processes in the system. The final section is a summary and discussion of AMPS.

9.1 Introduction

The previous chapters discuss modelling and planning for adaptive agents. As Chapter 5 explains, modelling in AMPS involves revising the mapping from states and actions to regions and estimating the parameters of an SMDP over these regions from experience. Chapter 6 describes approaches for introducing the necessary distinctions in the state and action space to produce competent behaviour. Chapter 8 describes how to incrementally plan over the model with prioritised value iteration. This chapter discusses the integration of modelling and planning in AMPS and its implementation as part of the Java Reinforcement Learning Framework (JRLF).

Although JRLF was developed specifically for the research presented in this thesis, the software package is likely to be of use to other researchers in testing and comparing their algorithms with AMPS. The complete source code is publicly available from `http://mykel.kochenderfer.com/jrlf` and may be distributed according to the GNU General Public License.

The discussion of JRLF and AMPS in particular in this chapter is at a high

level, providing an overview of the fundamental ideas in their design. The purpose is not to document the present software implementation but to explain the general structure and mechanism of the system. When designing a complex piece of software, one may employ any number of abstractions to achieve the same effect. However, some abstractions are better than others. A good abstraction is one that makes it easy to solve the problem at hand. Although there are general design patterns and methodologies a software engineer may follow (Gamma et al., 1995), it is not always obvious how to choose a suitable abstraction for a problem. JRLF and AMPS have incorporated several significantly different abstractions during their evolution. This chapter presents what appears to be the best abstraction used for their implementation.

9.2 General Framework

JRLF is written in Java 5 (Gosling et al., 2005; Arnold et al., 2006), an object-oriented, general-purpose programming language. The Java language was chosen because of its language features, the availability of software libraries, and its portability to multiple platforms. The programming language C# is another suitable language, but cross-platform support is still maturing. There are many publicly available reinforcement learning packages, but most assume discrete time steps and are implemented in C or C++. In addition, most existing packages make assumptions about the representation of the states and actions.

The purpose of JRLF is to provide a general framework for implementing and testing reinforcement learning algorithms in a variety of environments. The JRLF packages may be divided into three categories:

- **Agent:** Contains the implementation for a variety of agents, including AMPS.
- **Environment:** Contains the implementation for a variety of worlds, such as Taxi World and Race Track World.
- **Common:** Contains routines and data structures useful to different kinds of agents and environments.

JRLF includes functionality for running batches of experiments and visualising the interaction of the agent with the environment. Because AMPS and other

learning systems utilise decision graphs and graphical model representations, JRLF includes functionality for displaying and manipulating graphs. The graph display routines are designed to handle graphs whose nodes are frequently split and merged online. The visualisations for AMPS utilise an animated layout system based on the spring embedding algorithm by Eades (1984). The animation aids in understanding the dynamics of the revision processes and debugging. However, because the system is designed to aid visual recognition of gradual changes in the graph, the layout at any particular instant in time can be relatively poor compared to its rendering by a static layout engine (Tollis et al., 1999). In order to display high-quality static graphs, JRLF integrates with the open source Graphviz collection of layout algorithms (Gansner and North, 2000). JRLF uses the Graphviz implementation of the hierarchial layout algorithm described by Gansner et al. (1993) to display decision graphs and the layout algorithm described by Kamada and Kawai (1989) to display state-transition models.

JRLF begins by reading an XML file that specifies which modules to use for the environment, agent, and display. The XML file also specifies the parameters of experiments, including the number of episodes, sample frequency, and episodic timeout. After instantiating the environment, agent, and display, JRLF adds the agent and display to the list of *listeners* to the events raised by the environment. The environment raises three different events when

- the episode begins with some initial observation,
- an action is executed for some duration, emitting new samples of the current observation, lump-sum reward, and reward rate, and
- the episode terminates naturally.

In JRLF, the samples of the underlying states are called observations because observations are a more general term appropriate for partially observable environments. However, all of the environments included with JRLF are fully observable, and so the observations can be referred to as states (Section 3.5).

After the agent is added as a listener to the environment, JRLF proceeds by running the specified number of episodes. Each episode begins by resetting the environment to a random initial state. JRLF queries the agent for the action to take and then passes the action to the environment to execute for some duration. The process of querying the agent and executing the chosen action repeats until

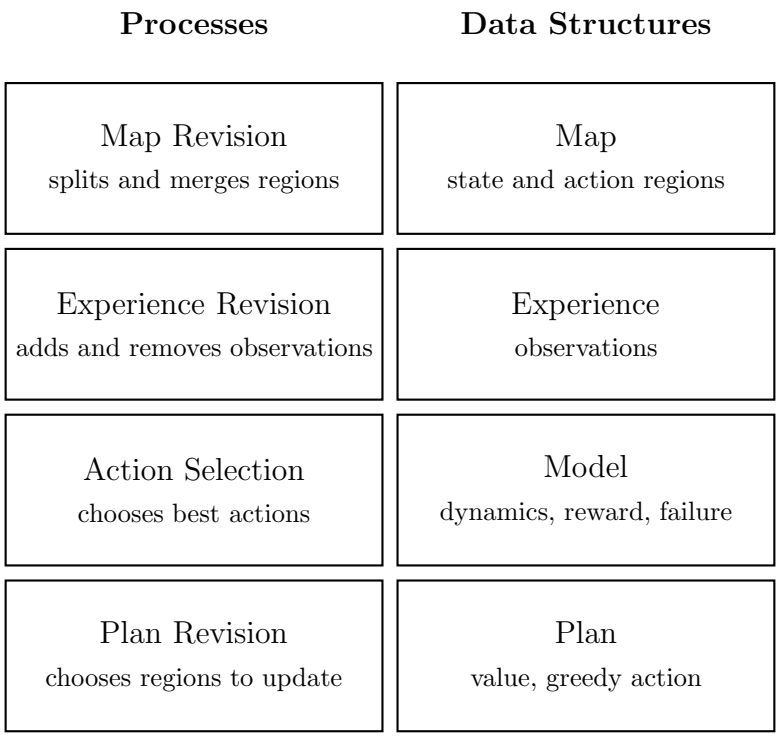


Figure 9.1: The processes and data structures in AMPS.

the total duration of the episode exceeds some specified value or the environment indicates that a terminal state has been reached.

The remainder of this chapter focuses on AMPS and its implementation as part of JRLF. AMPS is composed of four data structures and four processes, as shown in Figure 9.1. The data structures include the map, experience, model, and plan. Changes in one data structure are propagated to the data structures below it in the diagram as the next section explains. The four processes include map revision, experience revision, action selection, and plan revision. These processes are independent of each other and can be interleaved in any order.

9.3 Data Structures

This section discusses the main data structures in AMPS, including the map, experience, model, and plan. These data structures are defined abstractly to allow flexibility in implementation. Each data structure is responsible for storing and maintaining the data associated with it. As the next section discusses, the processes read from and perform operations on these data structures.

9.3.1 Map

The map data structure is responsible for mapping states to state regions and actions to action regions. The mechanism for mapping states to state regions may be different from the mapping of actions to action regions. For example, a nearest neighbour approach might partition the state space, and a decision graph might partition the action space. The map associates with each state region a separate mapping of actions to action regions.

The map supports splitting state and action regions according to sets of examples. The process involved in splitting regions follows the description in Chapter 6. The map also supports merging state and action regions as Chapter 7 explains. When the agent splits and merges regions, the map notifies the experience data structure of the change.

9.3.2 Experience

The experience data structure is responsible for storing and managing the experience of the agent. The data structure consists of a linked list of experience data, including samples of states, actions, durations, rewards, and indications of failure and termination (Figure 9.2). The experience data structure is also responsible for associating the experience with state and action regions in a way that is consistent with the map data structure.

When the experience data structure receives notification that the agent has acquired new experience, it adds the new experience to its linked list and records the region to which the sampled state and action belongs. If the map data structure splits or merges regions, the experience data structure updates the association between experiences and regions. The experience data structure notifies the model of any new experiences or changes in their association with regions.

9.3.3 Model

The model is responsible for maintaining an SMDP estimated from experience. The model listens for changes to the experience data structure and updates its estimate of the system dynamics accordingly. The model may use parametric or non-parametric estimation (Section 5.3).

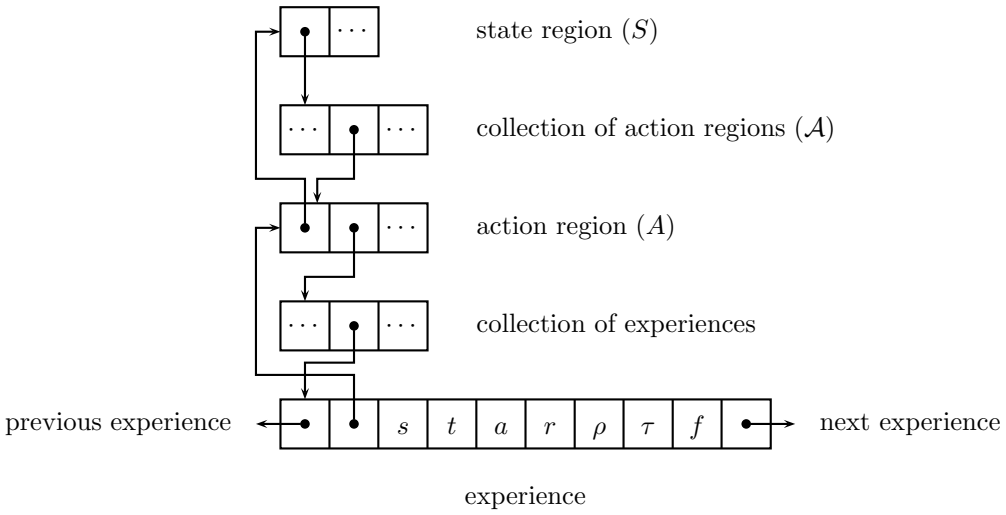


Figure 9.2: The organisation of the experience data structure. Each experience contains samples of the state s, transition duration t, action a, lump-sum reward r, reward rate ρ, termination τ, and failure f. Experiences are associated with action regions. The diagram also illustrates the one-to-many relationship between the state regions and action regions as maintained by the map data structure.

9.3.4 Plan

The plan data structure is responsible for maintaining the estimated value function V and optimal policy π. It associates $V(S)$ and $\pi(S)$ with S. The plan data structure is responsible for updating these values in response to requests from the plan-revision process. When a region is split, the plan data structure automatically performs value clipping (Section 6.6) to improve its estimate of V.

9.4 Processes

This section describes the four processes in AMPS. These processes perform operations on the data structures in the previous section. These four processes are independent of each other and may run in any order.

9.4.1 Map Revision

The map-revision process is responsible for requesting the map data structure to split and merge regions of the state and action spaces. Associated with the map-

revision process is a collection of revisors, such as the value revisor (Section 6.2), failure revisor (Section 6.2), and simplification revisor (Section 7.2). Each revisor is capable of computing the priority of its revision at a particular state region as well as performing the revision.

The map-revision process associates with each state region the highest priority revision and maintains a priority queue consisting of the regions whose priority is above a certain threshold. The revision process listens for changes in the data structures that impact the priorities it assigns to regions and updates the priorities accordingly. Since items are frequently added to and removed from a potentially large queue, it is important to have an efficient priority queue implementation such as a relaxed heap (Driscoll et al., 1988) or Fibonacci heap (Fredman and Tarjan, 1987).

When the agent has a small slice of time available, it can activate the map-revision process. The revision process quickly performs the highest priority revision at the highest priority state and returns control back to the agent.

9.4.2 Experience Revision

The experience-revision process is responsible for notifying the experience data structure when the agent acquires new experience. If the agent samples the state at a high frequency, it is not practical to record every experience. The experience-revision process can function as a filter, only recording experiences that it deems significant. The exact mechanism for determining significance is domain dependent, but one approach to filtering out states is to ignore all states until the agent has transitioned to a new state that is outwith some threshold distance.

In addition to filtering out insignificant experiences, the experience-revision process is also responsible for removing old experiences when memory and processor constraints deem it necessary. The removal of old samples allows the agent to adapt to slowly changing environments. Sometimes the removal of old experience may leave one or more of the state regions without samples. AMPS may remove any state regions without samples and revise the map data structure appropriately.

9.4.3 Action Selection

The action-selection process is responsible for continuously selecting a single action to execute. Action selection needs to be extremely efficient because it is typically executed as frequently as the agent samples the state of the world. A good action-selection strategy adheres to the following four principles:

- **The strategy should encourage taking an action consistent with the estimated optimal policy.** The purpose of planning is to identify the actions that are likely to maximise the expected return. The action-selection strategy should exploit the beliefs it obtains through planning by executing the actions likely to be valuable.

- **The strategy should encourage occasional exploration.** Although the agent should generally take the action recommended by the planning process, random exploration is also necessary in order for the agent to estimate an accurate model of the system dynamics.

- **The strategy should encourage taking the action taken in the previous step.** The continued application of an action is often necessary to transition from one abstract state region to another. If actions are frequently interrupted before completion, then the agent is not able to effectively estimate the parameters of the state-transition model.

- **The strategy should encourage executing actions that have not led previously to failure.** Failure indicates that the continued application of the same action is not likely to result in a useful transition. The action-selection strategy should avoid recommending actions that have led to failure.

The challenge is to balance these four competing principles. Most reinforcement learning algorithms attempt to balance the first two principles involving exploitation and exploration, however most algorithms in the literature are not designed to address the last two principles.

Algorithm 14 lists the basic action-selection strategy in AMPS. It follows the four principles of a good action-selection algorithm and is relatively simple. With probability inversely related to the estimated probability of failure, the agent will execute the greedy action. Otherwise, it will continue with the action it was

Algorithm 14 CHOOSE-ACTION(s)

$a \leftarrow$ last action taken
$S \leftarrow S(s)$
$A \leftarrow A(a)$
if RANDOM$(1 - \epsilon - (1 - 2\epsilon)P_f(S, \pi(S)))$ **then**
  if $A = \pi(S)$ **then**
    return a;
  return a random action in $\pi(S)$
if $A \neq$ NIL and RANDOM$(1 - \epsilon - (1 - 2\epsilon)P_f(S, A))$ **then**
  return a
return a random action

previously executing, again with probability inversely related to the estimated probability of failure. Otherwise, it will execute a random action. Random exploration is controlled by means of the parameter ϵ.

In Algorithm 14, the function RANDOM(p) is true with probability p and is false otherwise. In the algorithm, the probability of taking a greedy action is given by

$$1 - \epsilon - (1 - 2\epsilon)P_f(S, \pi(S)).$$

The selection probability above has two desirable properties. First, it decreases monotonically with respect to the estimated probability of failure and regardless of the failure rate. Second, regardless of failure rate, it is never deterministic, thereby allowing exploration. The probability of continuing the previous action is of a similar form.

The action-selection strategy in Algorithm 14 has the property that it will continue executing a greedy action within a state region for a duration selected from a geometric distribution, assuming a uniform sampling rate. As the sampling rate goes to infinity, the distribution over durations approaches an exponential distribution.

If value clipping (Section 6.6) is incorporated into the system, the action-selection algorithm should be adapted. If the value of a state S is zero, assuming that reward is always positive, then the best action to take from S is unclear and $\pi(S)$ in Algorithm 14 is not useful. In this case, the first *if* block in Algorithm 14 is ignored.

If an oracle is available for the agent to query (Section 5.5.2), then AMPS asks for assistance whenever the value of the current region is zero. If an oracle is not available, the agent simply follows Algorithm 14, taking a random action and continuing with that action at each step with some probability inversely related to the estimated probability of failure.

9.4.4 Plan Revision

The plan-revision process is responsible for updating estimates of the value function and optimal policy. AMPS uses prioritised value iteration to choose which state regions to update. As Chapter 8 explains, there are different ways of implementing prioritised value iteration. By default, AMPS uses an implementation that follows that of Moore and Atkeson (1993), but the other approaches are suitable for plan revision as well.

The plan-revision process listens for changes in the model, including the splitting and merging of regions, and for changes in the estimated value function as a result of value clipping. Depending on the nature of the changes, the plan-revision process updates the appropriate revision priorities. When the agent has time to perform planning, the planning process updates the value function and plan at the region at the front of the priority queue.

9.5 Discussion

In order for an agent to behave competently in an environment whose system dynamics are initially unknown, the agent must incrementally construct a model that generalises from experience and plan over this model to accomplish its objective. The previous chapters discuss various aspects of modelling and planning in isolation. This chapter discusses the integration of modelling and planning in an implemented system.

The beginning of this chapter introduces JRLF, the general software framework upon which AMPS is implemented. JRLF is an open source package designed to serve as a general reinforcement learning test bed, facilitating the side-by-side comparison of different learning algorithms in different environments. Although other reinforcement learning packages exist, JRLF is the only one publicly available in Java that allows variable-length time steps and flexible representa-

tions of the state and action spaces. It also uses XML to configure the various agent, environment, display, and system parameters. JRLF also integrates with other high-quality open source packages including the graph layout package Graphviz and the theorem prover JTP.

This chapter provides a high-level summary of the main data structures and processes in AMPS. Further details regarding the implementation are in the software documentation distributed with the source code. The various processes read from and adapt the data structures. This chapter explains how the various system components update themselves in response to events such as changes in the map or acquisitions of experience. AMPS was carefully designed so that it only spends time performing the essential updates. Because estimating the model and computing the plan is prohibitively expensive to do from scratch with every change in the abstraction, the system only performs local computation at regions affected by the change. Local revision is essential to an agent that adaptively revises its model in real time.

AMPS prioritises its modelling and planning operations. The modelling and planning operations involve quick, atomic updates of the data structures. Prioritising these updates allows the system to quickly perform the most needed updates when processing time becomes available. The agent may perform map-revision and plan-revision updates as frequently as the agent desires while sensing the world and making control decisions. The prioritisation approach in AMPS is especially useful for mobile robots with limited computing power and for virtual agents in computer games where only a fraction of the processor cycles may be spent on artificial intelligence routines (Khoo and Zubek, 2002).

Chapter 10

Evaluation

This chapter evaluates AMPS as a system for learning intelligent behaviour through interaction. After a brief introduction, the first part of this chapter demonstrates AMPS in a number of different environments and discusses the adaptation of the model and plan over time. The second part of this chapter closely examines the behaviour of the various components in the system. The final part of this chapter compares AMPS to several alternative approaches.

10.1 Introduction

The previous chapter shows how the ideas from Chapters 5–8 can function together to learn intelligent behaviour through interaction. The purpose of this chapter is to evaluate this system, to see how it performs on problems, how its components contribute to the behaviour of the agent, and how it compares to other approaches. The implementation for all of the experiments in this chapter is available as part of the Java Reinforcement Learning Framework (JRLF) as introduced in Section 9.2.

The next section demonstrates AMPS on three problems. Its purpose is primarily pedagogical; it does not aim to prove anything about the performance of AMPS in general. It contains several diagrams and plots illustrating the behaviour of the learning system over time. The plots in this section are of single runs over many episodes, and therefore do not contain error bars. Different random seeds, which govern the initial states at the beginning of episodes and the stochastic environmental dynamics, can result in different behaviour, but the kind of behaviour demonstrated in these plots is typical.

Section 10.3 examines individual components of AMPS and their influence on performance, and Section 10.4 compares AMPS to other methods including behavioural cloning, temporal-difference learning, prioritised sweeping, UTree, and TTree. Both sections involve experimentation and analysis. In general, the experiments use the same parameter settings as Section 10.2.

There are many ways to measure performance. Since the objective of this thesis is to study methods for learning *competent* behaviour—not necessarily *optimal* behaviour—it is necessary to have some definition of competence. Other researchers (e.g., Goodrich et al., 2000; Crook, 2006) have investigated learning *satisficing* behaviour instead of optimal behaviour. Simon (1956) coined the word *satisficing* in the context of agents with bounded rationality, using the word to mean behaviour that satisfies some minimum level of competency in achieving a goal. This chapter measures performance by counting the number of problems solved within a fixed time limit per episode. The experiments limit episodes to 500 steps in the Taxi World problem and 100 s of simulated time in the Corner World problem. These limits accommodate a small amount of random exploration in each episode, regardless of initial state.

Many of the experiments compare estimates of mean performance, run time, and model complexity. When reporting estimates of the mean of a random variable, this chapter always provides 99% confidence intervals in both the text and in diagrams with error bars. Estimates in this chapter are always based on 100 samples to allow for reasonably tight confidence intervals.

Some of the experiments involve comparing the performance of different algorithms. Claims that one algorithm performs better than another on a particular problem are based on 100 runs with typically 100 episodes in each run to allow the behaviour of the agent to stabilise. The degree of significance of these claims is measured by the Wilcoxon signed-rank test (Wilcoxon, 1945) and p-values are provided in the text. This chapter uses the Wilcoxon test because the data points are paired as a result of using the same series of initial states across runs.

10.2 Demonstration

This section demonstrates AMPS in the Goal World, Taxi World, and Corner World domains. This section will begin with a discussion of the Goal World domain because it is the simplest to understand and for AMPS to solve.

10.2.1 Goal World

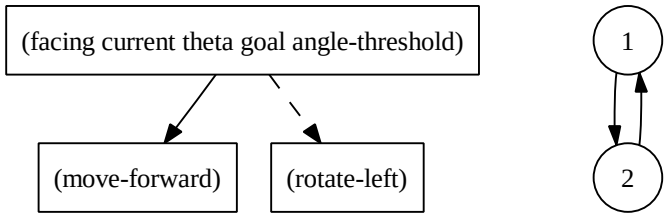
As Section 1.4.1 explains, the Goal World domain involves a mobile robot that must maneuver to reach some goal without any prior knowledge of the system dynamics (see Figure 1.2 on page 6). The state space is a subset of $\mathbb{R}^5$ and the agent is able to observe its position and orientation and the absolute position of the goal. The robot may move forward and rotate left and right at some fixed velocity.

In order to build an abstract SMDP, AMPS needs some way to partition the state space. As Chapter 6 discusses, there are many ways to represent the partitioning of the state space. For example, one might use a nearest neighbour approach to cluster together similar states according to a distance metric, or one might use a decision graph that partitions the state space through a series of tests.

If AMPS is to use a decision graph to partition the state space, it must be provided with a description of the kinds of tests it is to use. Since the state space can be represented as a real-valued vector, one might wish to use hyperplanes to carve the state space into a finite set of regions. Alternatively, one could use grounded predicates as tests. The current implementation of AMPS supports the specification of typed relations, functions, and constants in XML. When AMPS discovers that it might be advantageous to introduce a distinction in the state space, it constructs a grounded predicate from the provided set of relations, functions, and constants that maximises some quality measure such as information gain (Section 6.4.1).

The set of types for the Goal World domain include states, points, angles, and scalar thresholds. There are several functions that take parameters and return values of these types. The relations include *facing*, *left*, *right*, and *close*. Since there are many ways of combining the relations, functions, and constants to form a grounded predicate, it is desirable to eliminate any unnecessary predicates from consideration, which is done using a set of theorems and a theorem prover as Section 6.4.4 explains.

Without a prior model of the dynamics and without intermediate reward to encourage progress towards the goal, the Goal World task is quite difficult. Random exploration by itself is unlikely to be successful within a reasonable amount of time. Therefore, the first episode involves an approximately optimal



(a) The map. (b) The model.

Figure 10.1: The map and the model learned in the Goal World domain. The map partitions the state space with a decision graph. This particular decision graph distinguishes states according to whether the robot is within some threshold of facing the goal. The model is represented by an SMDP estimated from experience. Shown is the connectivity between the two regions.

teacher steering the robot to its goal. The initial state is selected randomly from a uniform distribution over all possible states. The duration of this training episode is random, but for the run described here it lasted 6.5 s with decisions being made at 10 Hz.

After this first episode, AMPS is left on its own to control the robot. At first, AMPS treats the entire state space as a single region. Because the action Move-Forward was the action that the teacher used immediately before reaching the goal, the planning process identifies Move-Forward as the greedy action. AMPS begins executing the action Move-Forward until it crashes into the wall. Since crashing into the wall is identified as a failure, the failure-revision process (Section 6.2.2) splits the state space based on samples that led to failure and samples that led to success. Figure 10.1 illustrates the state of the decision graph and the model after the split.

After the state space is split, AMPS performs prioritised plan revision (Chapter 8) and arrives at a competent plan. The plan is to rotate left until the robot faces the goal and then move forward. If moving forward results in the robot not facing the goal, then the robot rotates left until it faces it again. This plan can be suboptimal because in some situations it may be better to rotate right instead of left. In this experiment, the agent was not provided with the necessary relations and functions to determine whether it is quicker to turn left or right to face the goal, so it is not surprising that the learned policy is suboptimal.

10.2.2 Taxi World

The Taxi World domain (Section 1.4.2) is much more complex than the Goal World domain. This task involves picking up passengers and depositing them at their destinations. The taxis, passengers, and destinations occupy single cells within a 20×20 grid (see Figure 1.3 on page 7). The dynamics are nondeterministic. With probability 0.1, the taxi moves in a direction orthogonal to the intended direction.

Again, it is necessary to define the representation for perceptual distinctions. As with the Goal World task, AMPS might use a decision graph with predicates synthesised from an XML file. Some of the binary relations provided include *north*, *south*, *east*, *west*, *directly-north*, *directly-south*, *directly-east*, and *directly-west*. A set of theorems specify when it is possible to prune grounded predicates from consideration (Section 6.4.4).

As with the goal-world domain, it is extremely unlikely that random exploration will result in the agent accomplishing its task in a reasonable amount of time. Therefore, a noisy teacher directs the agent through the task. The noisy teacher chooses a random action 10% of the time and an optimal action the remainder of the time. In this chapter, the Taxi World experiments involve three training episodes by this noisy teacher. Three training episodes appears to be just enough training to enable the agent to solve the problem without having to rely too heavily on random exploration. The number of control examples provided by the noisy teacher in three episodes depends on the random state initialisation, the stochasticity in the system, and control noise. In this initial run, the teacher makes 126 control decisions over the three training episodes.

Following the three training episodes, AMPS is left on its own. Its initial simplistic model of the system dynamics is not sufficient for solving the task. The agent begins by trying to put down a passenger that is not in the taxi, resulting in failure. The map-revision process adds a perceptual distinction and the plan-revision process updates the plan. The agent then moves left until it crashes into the wall, resulting in failure and the addition of another perceptual distinction. It makes a few more mistakes until it heads towards the passenger. By the time the taxi reaches the passenger, the state space is divided into seven regions. The taxi then successfully picks up the passenger and delivers the passenger to its destination. Figure 10.2 shows the structure of the SMDP after the successful

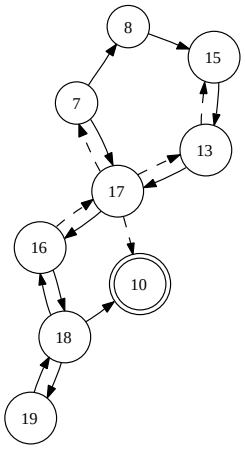
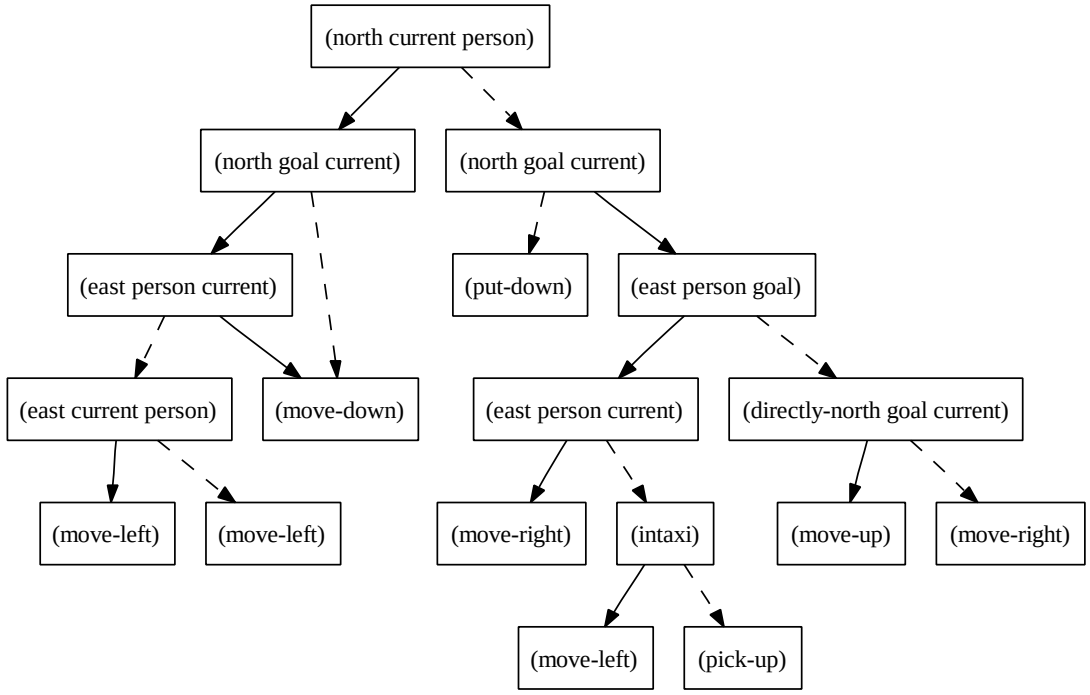


Figure 10.2: The model learned in the Taxi World domain after one episode. Greedy transitions in the model are indicated by solid directed edges and non-greedy transitions are indicated by dashed directed edges. Region 10 (indicated with a double border) contains the goal.

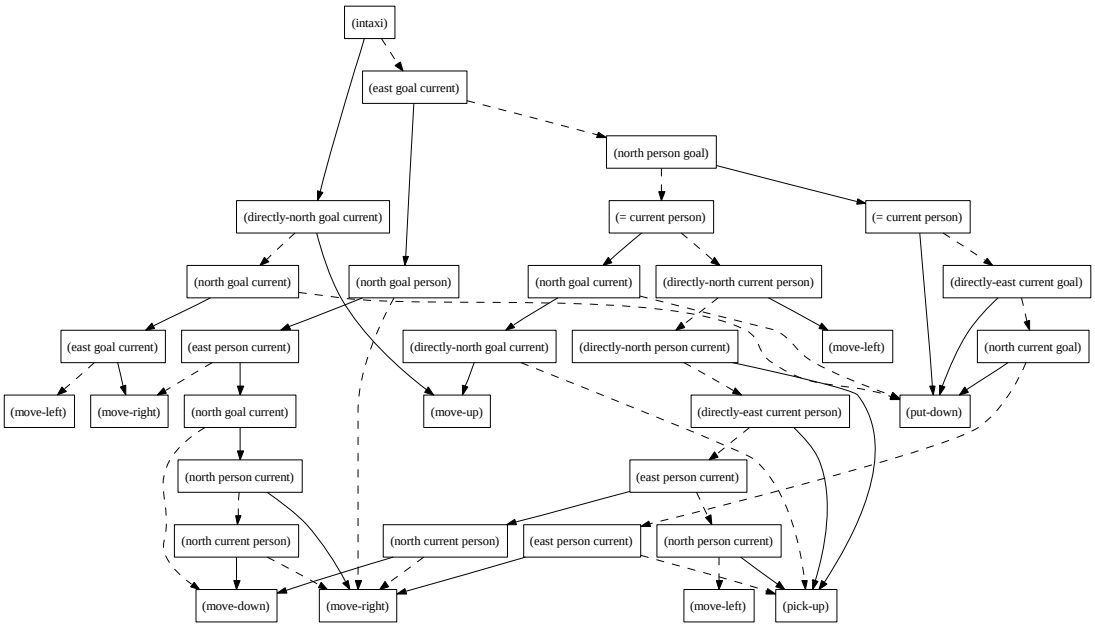
completion of the task. At the end of the episode, AMPS reconstructs the map (Section 6.4.4). Figure 10.3 shows the map before and after the reconstruction. At this stage, reconstruction results in a more complex decision graph.

Following two successful episodes on its own, AMPS struggles in the sixth episode and does not manage to complete the task successfully within 500 steps. The failure of AMPS in this episode is due to "thrashing" between regions 55 and 52 (Figure 10.4). From region 55, the agent tries to move right so that it can transition to region 53 on its way to region 42, the region containing the goal. However, moving right can also bring the agent to region 52. Because of the actual position the taxi is in at this point in the episode, moving right will usually bring the agent to region 52 instead of region 53. Once in region 52, the agent tries to move left, bringing it back to region 55, and so the cycle continues until the agent takes an exploratory action or revises its model. Thrashing in this situation is due to perceptual aliasing in region 55. Eventually, the transition-revision process makes a perceptual distinction to resolve the issue with aliasing when the priority of adding that distinction rises above a specified threshold (Section 9.4).

The agent is able to solve the task within the allotted time for 88 of the 100 episodes. Figure 10.5 shows the structure of the decision graph at the beginning of the tenth episode.



(a) The map before reconstruction.



(b) The map after reconstruction.

Figure 10.3: The map learned in the Taxi World before and after reconstruction. As can be seen, the complexity of the map increases after the reconstruction.

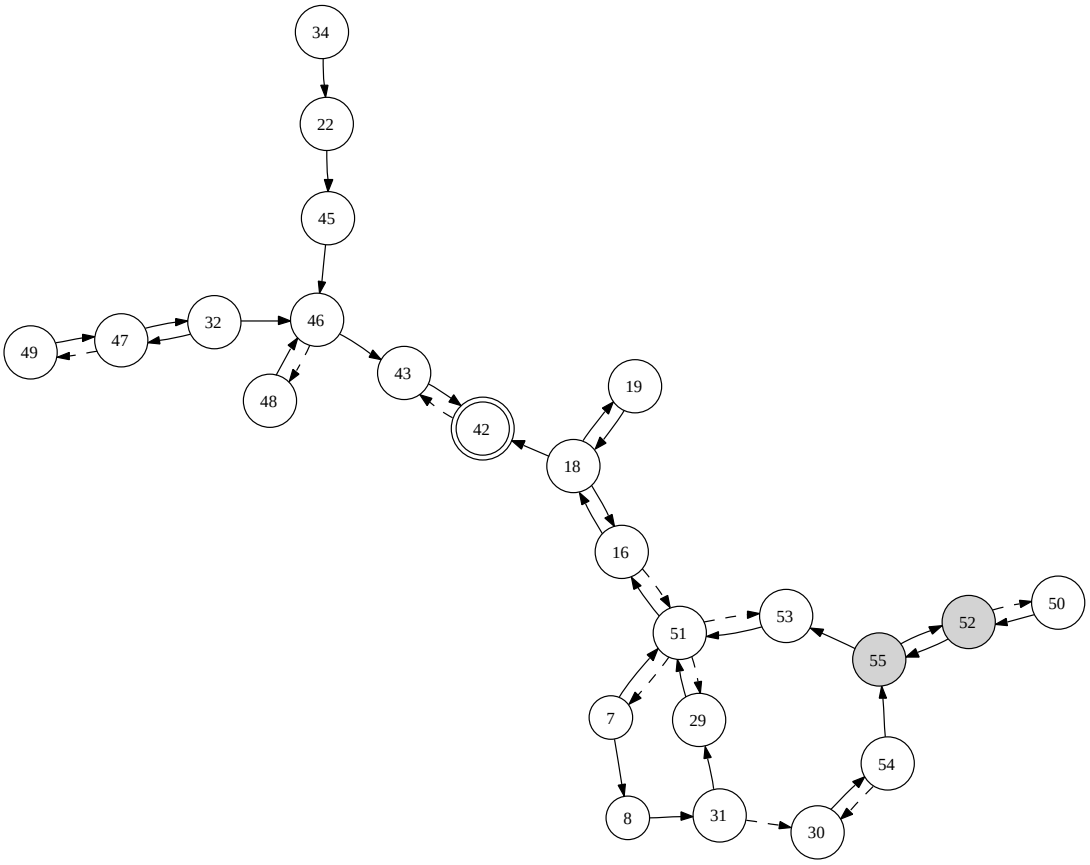


Figure 10.4: The Taxi World model in the sixth episode where thrashing occurs between regions 55 and 52 (shaded). Region 42 (indicated with a double border) contains the goal.

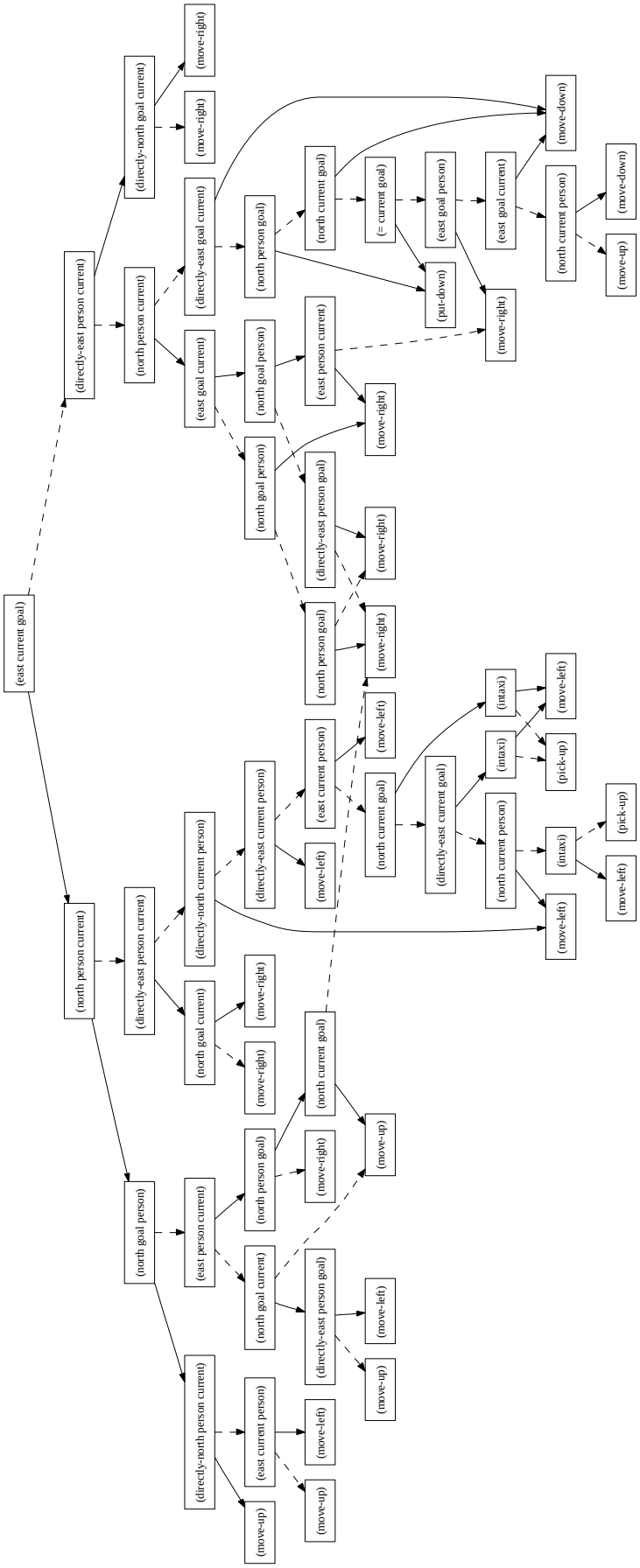


Figure 10.5: A decision graph learned while interacting with the Taxi World domain.

10.2.3 Corner World

The Corner World domain (Section 1.4.3) involves a holonomic robot situated in a two dimensional world. The agent begins at a starting line and the task involves navigating around a sharp corner and then heading towards the finish line (see Figure 1.4 on page 7). In contrast with the Taxi World domain, there are uncountably many states and actions. Hence, the agent must be capable of generalisation in both the perceptual and actional spaces. This section describes two ways of achieving generalisation in this problem. The first involves learning a decision graph, and the second involves instance-based generalisation with a distance metric.

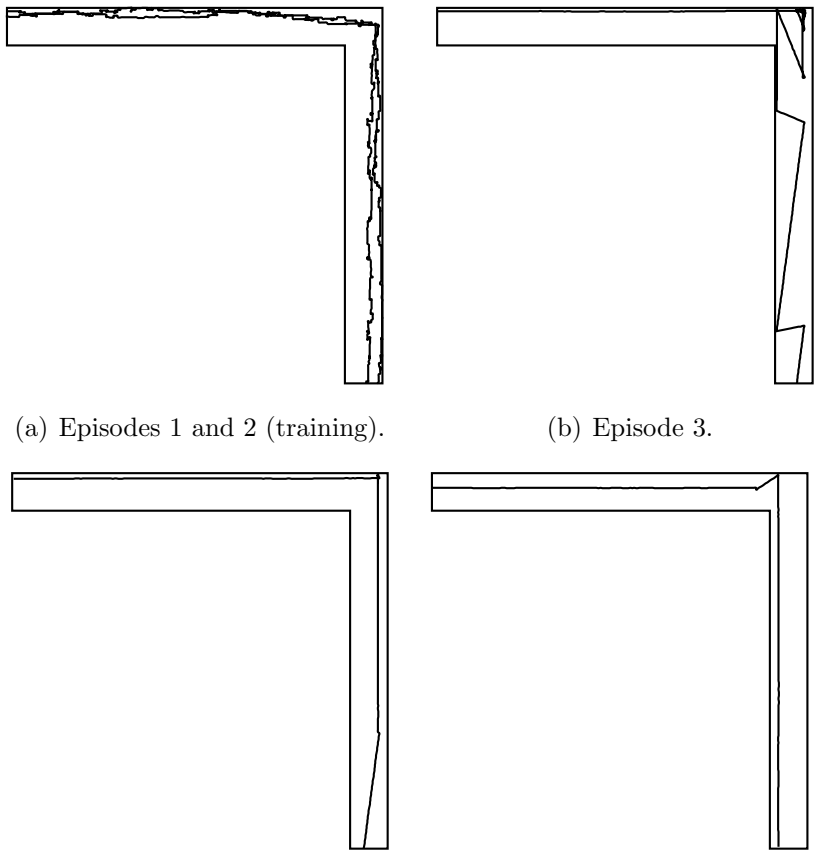
When using a decision graph for generalisation in the Corner World domain, it is useful to make distinctions using hyperplanes. To avoid overfitting and to reduce computation, AMPS only considers distinctions involving basis-orthogonal hyperplanes (Section 6.4.3). The tests that partition the state space involve comparing either the x or y coordinate of the position of the agent with some constant value. The tests that partition the action space involve checking whether the direction of the movement of the agent falls within some open interval.

As with the other domains in this section, random exploration is not likely to bring the agent to the goal. In this chapter, the agent is provided with two training episodes by a noisy teacher. A quarter of the time the teacher performs random actions and the remainder of the time the teacher follows a competent, but suboptimal, hand-crafted policy. Figure 10.6(a) shows the trajectories of the two training episodes.

On the third episode, AMPS continues on its own, revising its model and plan as it acquires experience. Since its model of the system dynamics is initially too simplistic, the agent at first has difficulty navigating towards the goal as shown in Figure 10.6(b). However, AMPS is able to revise its model and plan quickly enough to complete its task within a reasonable amount of time.

Figure 10.7 shows the model and the decision graph that partitions the state space that AMPS learns by the end of the third episode. The figure also shows the partitioning of the action space associated with one of the state regions. The model AMPS learns by the end of the third episode provides a smooth trajectory to the goal in the fourth episode as Figure 10.6(c) shows.

As Section 9.4 discusses, the experience-revision process purges old experi-



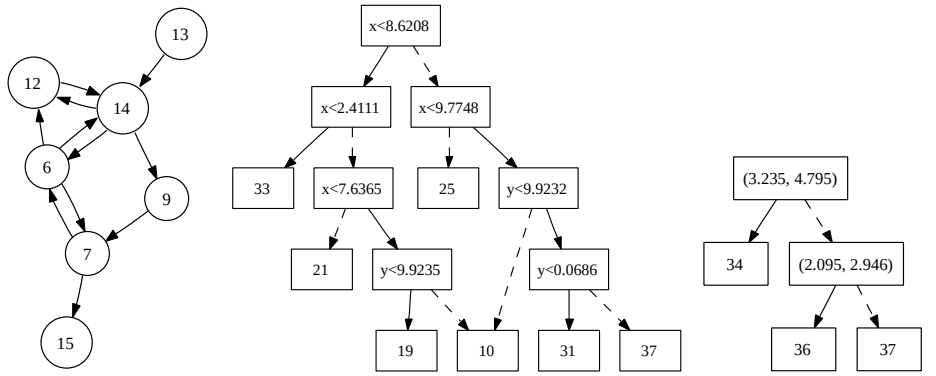
(a) Episodes 1 and 2 (training).

(b) Episode 3.

(c) Episode 4.

(d) Episode 100.

Figure 10.6: Trajectories through Corner World at various stages when using AMPS with decision graphs that partition the state and action spaces.



(a) The model. (b) The state map. (c) The action map.

Figure 10.7: The model and the state and action maps learned by the end of the third episode in the Corner World. Each node in the model is labelled by the ID of its associated state region. Each leaf node in the state map is labelled by the ID of the greedy action region associated with that region of the state space. Each leaf node in the action map is labelled by the ID of the action region. The action map shown is associated with the state region with ID 14.

ences. In this sample run, AMPS only retains experience from the ten most recent episodes in memory. When the agent starts the eleventh episode, AMPS purges the experience from the first episode. Keeping only ten episodes worth of memory for this task does not impair performance. More complicated tasks might require retaining more experience in memory.

Figure 10.6(d) shows the trajectory of the agent in its 100th episode. The trajectory is close to optimal, but the agent could have started moving left sooner around the corner. Even with more experience, the trajectory is still unlikely to be exactly optimal because of the need for explorative actions and the tradeoff between simplicity and accuracy in the model.

During the 100 episodes, AMPS revises the model by splitting and merging regions. As Figure 10.8 shows, the first few episodes involve increasing the complexity of the model. AMPS adds perceptual distinctions because of observed differences in expected value and failure along trajectories. The complexity of the model stabilises and with additional experience it becomes apparent to AMPS that it can simplify the model by merging some of the regions.

This initial simple model learned by AMPS produces reasonable behaviour until the 29th episode. The agent crashes against the walls several times in

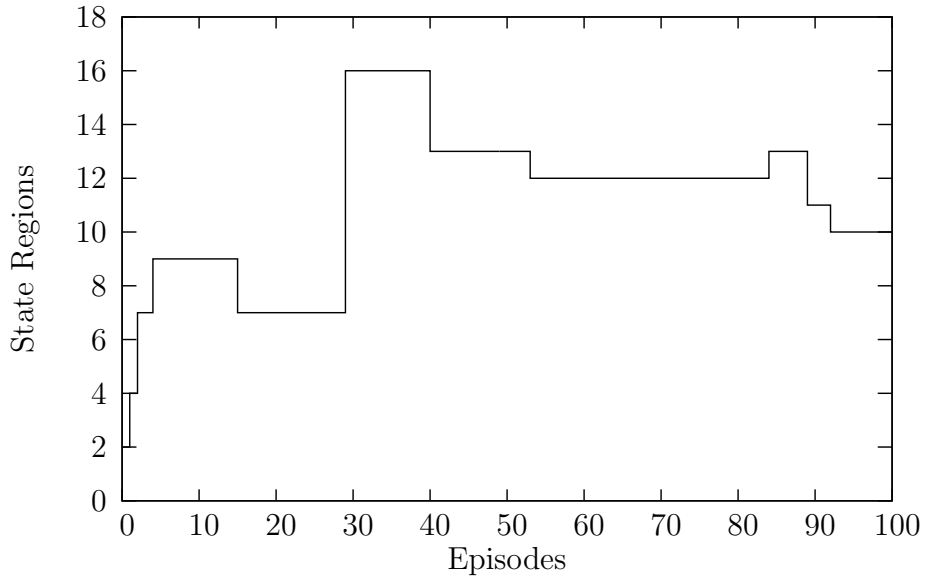


Figure 10.8: A plot showing how the number of state regions in the model changes as the agent acquires experience in the Corner World domain. As the agent begins to acquire experience, there is a steep increase in the number of state regions. Eventual reductions in the number of regions are due to the simplification process.

this episode. Although the learned model is good at predicting the dynamics of the system when the agent glides through the middle area of the track, the model does not accurately represent the dynamics when the agent is close to the wall, leading to relatively poor behaviour in episode 29 as shown in the plot in Figure 10.9. By the end of the 30th episode, the number of state regions doubles to accommodate the complexity of modelling wall interactions. With additional data, AMPS eventually simplifies the model. The behaviour during the 100 episodes is usually close to optimal and better than that of the teacher (Figure 10.9). By the end of the 100th episode, there are only ten state regions that partition the state space as shown in Figure 10.10.

Instance-based generalisation with a distance metric also provides good results. As Figure 10.11 reveals, AMPS typically solves the task 20% faster than the noisy teacher. Only twice in the 100 episodes of the sample run does AMPS solve the task slower than the teacher. As Figure 10.12 shows, nearest neighbour generalisation partitions the state space differently from decision graph generalisation. Figure 10.13 illustrates the variation of model complexity over time.

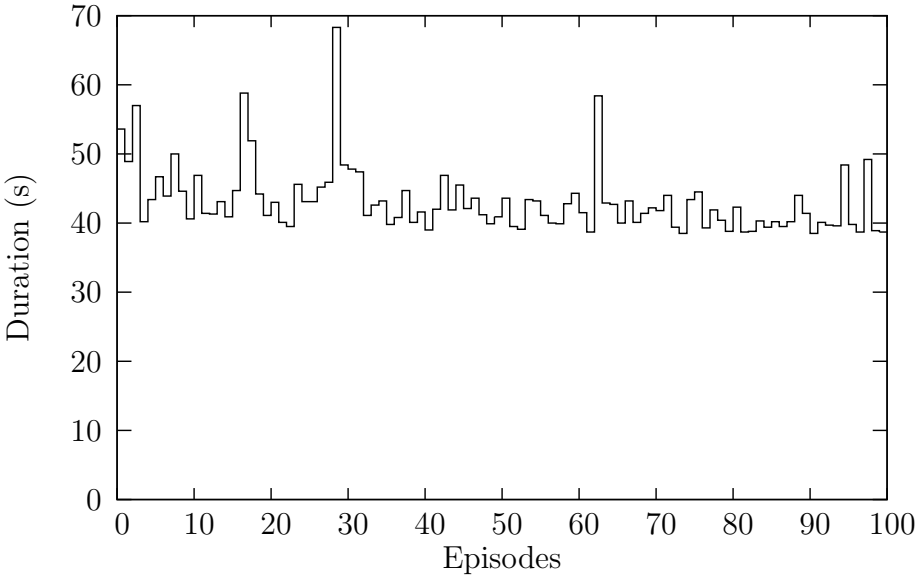


Figure 10.9: A plot of the duration of time required to solve the Corner World task over a series of episodes. Variation is due to randomness in the initial position on the starting line, environmental stochasticity, exploratory actions, and the evolution of the model and plan.

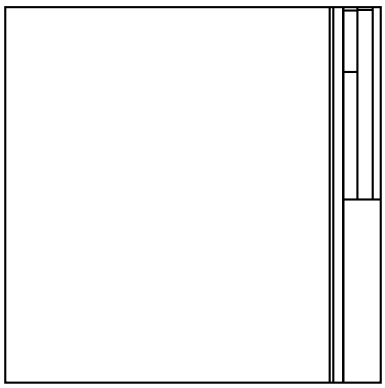


Figure 10.10: A physical partition of the state space at the end of the 100th episode of Corner World. There are ten regions. One region is extremely narrow and difficult to see at this scale. The two rectangular slivers near the top right of the space correspond to the same region.

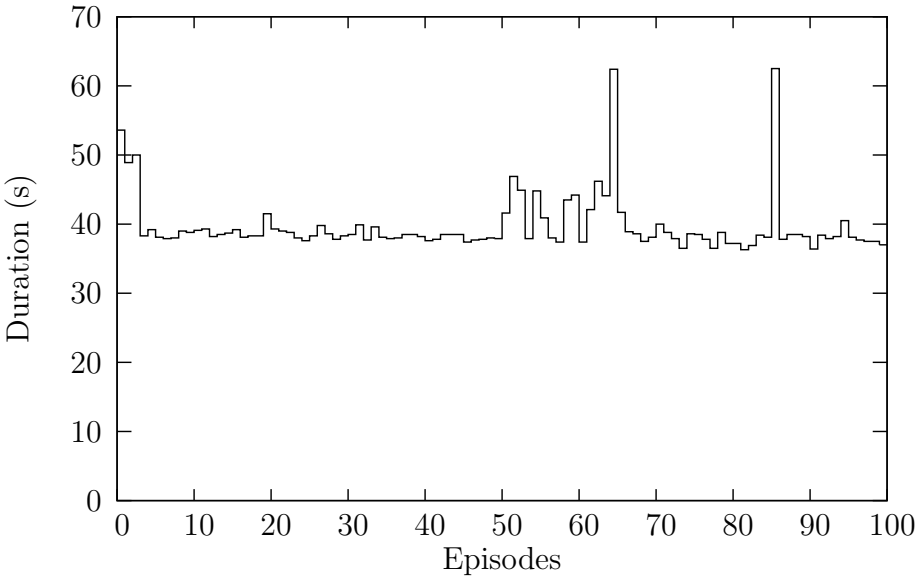
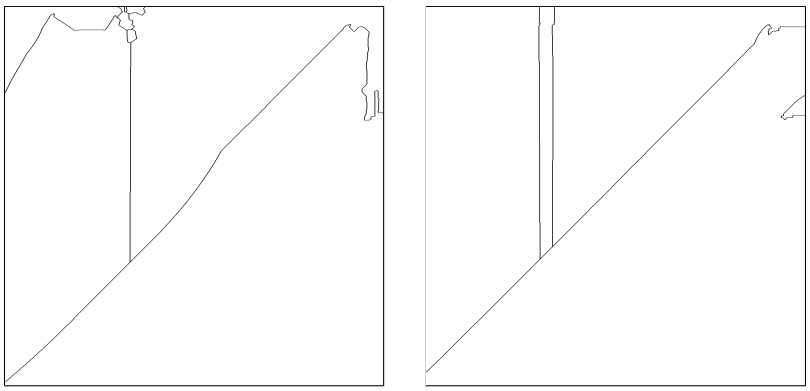


Figure 10.11: A plot of the duration of time required to solve the Corner World task over a series of episodes using nearest neighbour generalisation. AMPS almost always performs significantly better than the teacher after only two training episodes.



(a) Episode 5. (b) Episode 50.

Figure 10.12: A physical partition of the Corner World state space at the end of episodes 5 and 50 learned with nearest neighbour generalisation.

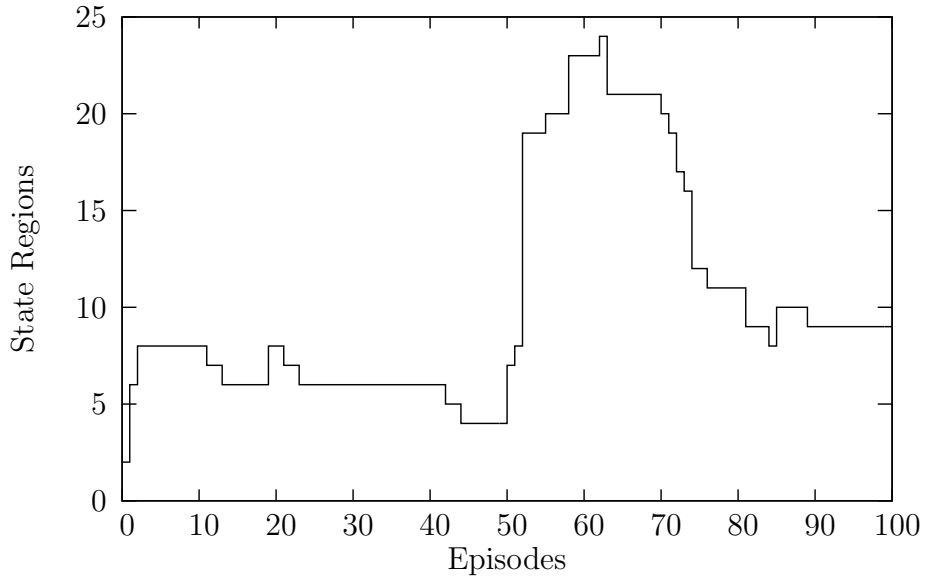


Figure 10.13: A plot showing how the number of state regions in the model changes over time using a nearest neighbour approach in Corner World.

10.3 Examination

The previous section provides a high-level demonstration of AMPS in various domains. This section examines the mechanisms involved in AMPS in greater detail. In particular, this section closely examines separation heuristics, model simplification, map simplification, value clipping, and planning. This section also examines the effect of batch revision on performance.

10.3.1 Separation Heuristics

In many problems, it is important that AMPS uses both the value-revision heuristic and the failure-revision heuristic (Section 6.2). The importance of these two heuristics interacting together is especially pronounced in the Taxi World domain. If AMPS is left on its own for ten episodes after three episodes of training (as in Section 10.2.2), the number of problems it solves when episodes are limited to 500 steps is

- 6.20 ± 0.66, when using both value and failure revision,
- 0.05 ± 0.06, when using only value revision, and
- 1.46 ± 0.50, when using only failure revision.

Value revision is useless without failure revision because it never results in a split. Value revision can only produce splits when there is a significant difference in value along different transitions leaving a state region in the model. When the agent commences its interaction with the world, there is only one state region other than the synthetic terminal region (Section 5.4.2), making it impossible for value revision to introduce any perceptual or actional distinctions.

Failure revision is always the first to introduce a split. Once the state or action space is split, the value-revision process is able to refine the model. For the Taxi World problem, value revision is essential for good performance. Failure revision by itself does not produce competent behaviour. However, in some domains such as Corner World, failure revision by itself may be enough to produce satisfactory behaviour.

10.3.2 Model Simplification

In addition to introducing perceptual and actional distinctions into the model, it is important for the agent to simplify the model when additional experience indicates that doing so would be useful (Section 7.2). Without simplification, the number of state and action regions can grow very quickly.

Figure 10.14 shows how the number of state regions changes over time with and without simplification in the Corner World domain. With simplification, the number of state regions in the model stays relatively small and constant. Without simplification, the number of state regions increases very quickly over time.[1] By the 100th episode, AMPS with simplification learns a model with only 10 regions and AMPS without simplification learns a model with 185 regions. Without model simplification, the agent solves only 58 problems instead of 100 and the simulation requires over three times the computation. In the Taxi World domain, the difference between AMPS with simplification and AMPS without simplification is less pronounced, but still significant as Figure 10.15 shows. For the experiments in this chapter, the agent retains 25 episodes of experience for the Taxi World domain and 10 episodes of experience for the Corner World domain.

The benefits of having a simpler model extend beyond savings in computation and memory. Fewer state and action regions can provide better generalisation

[1] Any decrease in the number of state regions without model simplification is due to the automatic removal of regions that are left without samples after the experience-revision process purges old experience (Section 9.4.2).

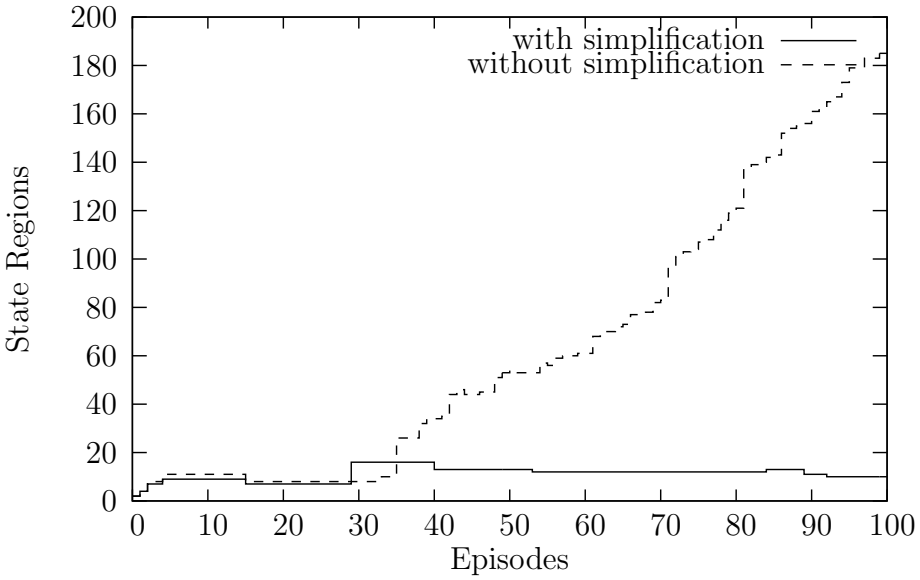


Figure 10.14: A plot showing how the number of state regions in the model changes over time using a decision graph approach with and without model simplification in the Corner World domain.

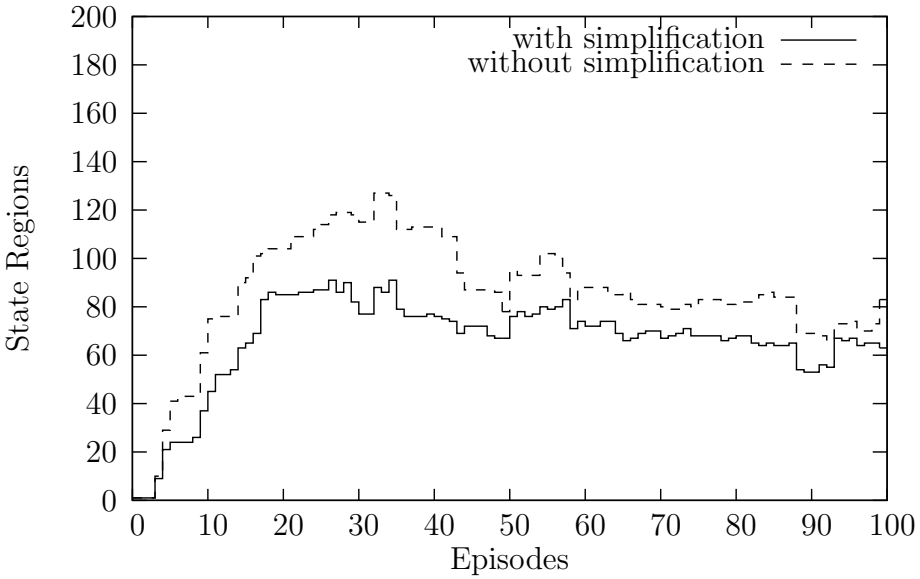


Figure 10.15: A plot showing how the number of state regions in the model changes over time using a decision graph approach with and without model simplification in the Taxi World domain.

from limited data. With fewer regions, more experience is available to estimate the transition probabilities, reward, and duration distribution. With a better model, the agent is able to construct a better plan. With model simplification, the agent is able to solve all 100 episodes of Corner World within a limit of 100 s of simulated time per episode. Without model simplification, the agent is only able to solve 58 episodes within the time limit. In the Taxi World domain, the agent solves 89 episodes with simplification and 82 without simplification.

10.3.3 Decision Graph Simplification

Section 7.3 discusses several complementary approaches to decision graph simplification. When using a decision graph to partition the state or action space, it is usually important to incorporate some sort of process that simplifies the structure of the graph. Without some kind of simplification process, the decision graph can quickly grow unnecessarily complex, resulting in inefficient classification and unnecessary utilisation of memory.

Figure 10.16 compares the effect of different decision graph simplification schemes on the number of nodes in the decision graph. Without any form of decision graph simplification as the agent acquires experience in the Taxi World domain, the number of decision nodes rapidly increases around the 25th episode. Removing redundant parents (rrp) significantly slows this growth. Removing redundant splits (rrs) does not have a significant impact on the number of decision nodes. Removing redundant splits only redirects directed edges in the graph to reduce the number of decisions necessary for classification; only rarely does this redirection result in removing decision nodes. Combining all simplification schemes along with decision graph reconstruction results in a stable number of decision nodes.

10.3.4 Value Clipping

When interleaving incremental modelling with planning, updating the values of state regions solely through prioritised value iteration (Section 8.3) is not sufficient for satisfactory performance. There are situations where prioritised value iteration takes unnecessarily long to converge. Section 6.6 introduces value clipping as a way to speed the process. Value clipping is an efficient process that clips value estimates that are noticeably too high or too low given a quick inspection

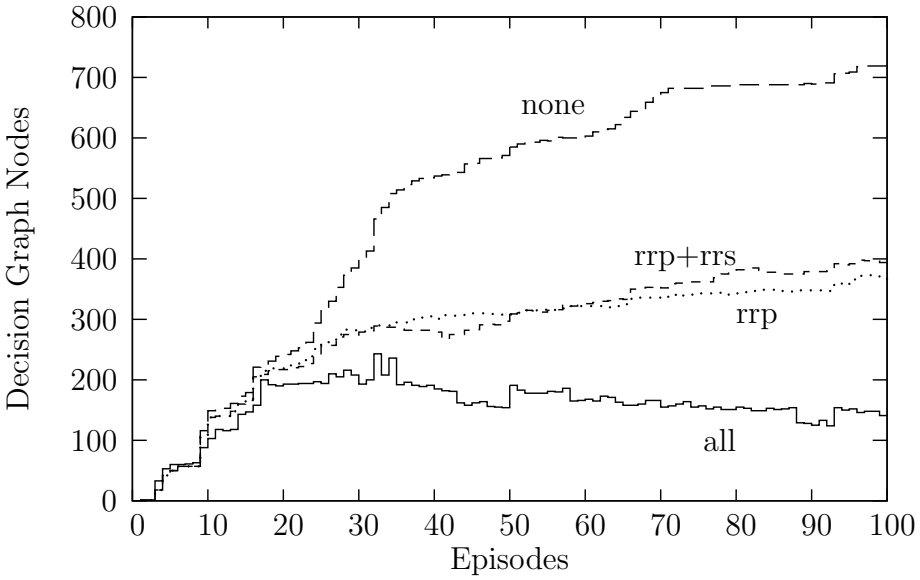


Figure 10.16: A plot showing how the number of decision graph nodes changes over time when using different simplification schemes including removing redundant parents (rrp) and removing redundant splits (rrs). This data comes from the Taxi World domain.

of the estimated reward function and model topology.

A sample run of 100 episodes of the Taxi World problem using the same parameters as usual illustrates the frequency of value clipping (Figure 10.17). Although there are some episodes that do not involve value clipping, some episodes require over 35 clips. Value clipping is most important during the early episodes when the experience of the agent in the environment is sparse. During the first 25 episodes without a teacher, AMPS with value clipping is expected to solve 18.48 ± 1.23 problems whereas AMPS without value clipping is expected to solve only 13.48 ± 1.17 problems. Not all types of problems stand to gain such a significant benefit from value clipping. Model revision in the Corner World domain, for example, rarely results in a need for value clipping because of the connectivity of the state space.

10.3.5 Planning

Chapter 8 discusses planning in isolation, ignoring issues of generalisation. One of the most interesting conclusions in that chapter is that if the agent is learning the model through interaction, the amount of planning and its prioritisation scheme

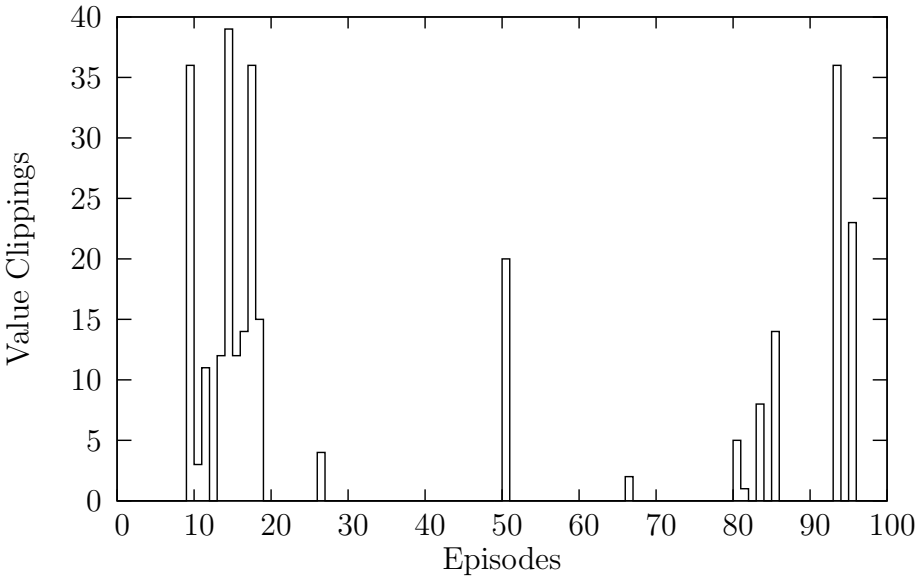


Figure 10.17: A plot of the number of clips performed by the value-clipping process over 100 episodes in the Taxi World domain.

has little impact on performance. As Table 8.2 in that chapter shows, even performing a single value update at the most recently visited state is sufficient for good performance, and the more sophisticated prioritisation schemes only marginally enhance performance. So long as the value of the most recently visited state is updated at every step, the main challenge is using experience effectively to construct a suitable model.

AMPS, by default, updates at every step the value function and plan at the most visited region. If AMPS does not update regions where the estimated model has changed, it performs extremely poorly. AMPS also uses value clipping by default to improve performance. Additional planning outside value clipping and performing updates at the most recent region is likely to be of only marginal benefit (see Figure 10.18). The main challenge in AMPS and other model-based algorithms is modelling, not planning.

10.3.6 Batch Revision

The complexity of most interesting problems makes it impractical to learn a new abstraction from scratch with each new experience. Hence, AMPS performs incremental model revision as described in this thesis. However, one may wonder what effect batch learning has on the behaviour of the agent, ignoring issues of

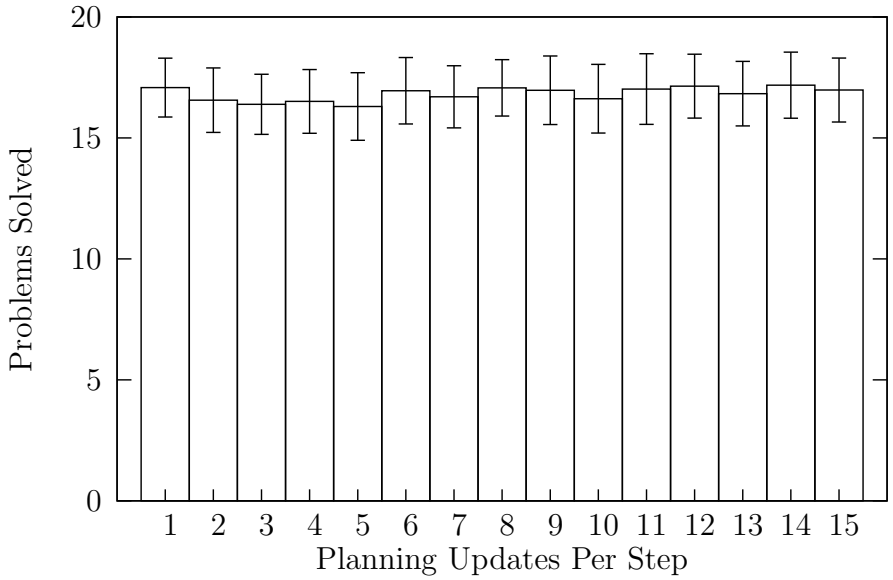


Figure 10.18: A histogram showing the expected number of problems solved for varying numbers of updates per step in Taxi World.

computation time.

To measure the effect of batch learning, an experiment was designed that involves two sets of 100 runs in the Taxi World domain. The first set involves running a version of AMPS through three training episodes, followed by ten episodes of AMPS on its own. After these learning episodes, the map is frozen, and the agent is tested on ten additional episodes. The second set of runs is identical, except that the map is induced from scratch before running the last ten episodes.

Somewhat surprisingly, the incremental version of AMPS outperformed the batch version of AMPS. In the experiments, the incremental version solved 14.03 ± 1.20 of the 20 problems, and the batch version solved only 8.46 ± 0.76. Similar results were found in the Corner World domain. Although the models induced by batch learning are generally simpler, the experiments show that they tend not to make enough perceptual and actional distinctions to do well on the tasks.

10.4 Comparison

This section compares AMPS with behavioural cloning, temporal-difference learning, prioritised sweeping, UTree, and TTree. Side-by-side empirical evaluation of

performance can be misleading. Different approaches make different assumptions about what is known. For example, prioritised sweeping assumes a discrete state and action space whereas AMPS learns discretisations while it interacts with the world. Behavioural cloning assumes the availability of a competent teacher, but temporal-difference learning can eventually learn satisfactory behaviour without any guidance from an external source. The purpose of this section is not to identify which approach is "best." After all, the best approach to use strongly depends on the prior knowledge of the agent, the structure of the learning process, and the constraints of the task. The purpose of this section is to identify the strengths and weaknesses of various approaches and how they relate to AMPS.

10.4.1 Behavioural Cloning

As Section 2.3 discusses, an agent may learn to solve a task by supervised learning. The transfer of expertise from a skilled teacher to a learning agent is known as behavioural cloning. There have been many successful applications of behavioural cloning, but there are some major limitations to this approach in general.

Perhaps the most fundamental problem with a behavioural clone is that it does not understand the effects of its actions, rendering it incapable of independent problem solving. Behavioural clones depend entirely on a skilled teacher. The agent is unable to use the experience it accumulates on its own to its advantage. The inability of the agent to adapt to novel situations or to changing dynamics is a severe limitation.

One may select from a variety of supervised learning methods to generalise from the training instances provided by a teacher. To make comparison with AMPS easier, this section focuses on decision graph and nearest neighbour methods for supervised learning.

Generalisation in the Taxi World domain is more natural using a decision graph than a distance metric. A supervised learning algorithm may use the same mechanism that AMPS uses in Section 10.2.2 to introduce distinctions in the state space. The learning algorithm may split the state space incrementally in a top-down fashion like other tree-induction systems (e.g., Quinlan, 1993). If there are multiple leaf nodes belonging to the same category, the algorithm may merge these leaves together. After three training episodes in Taxi World, a decision graph induction system learns the decision graph shown in Figure 10.19. Learning

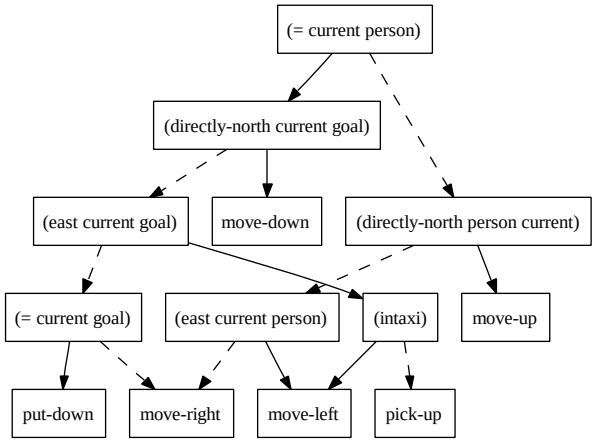


Figure 10.19: A decision graph learned by a behavioural clone for the Taxi World domain.

	AMPS	clone
decision graph	70.04±8.37	55.81±6.26
nearest neighbour	95.42±3.46	54.33±8.57

Table 10.1: A comparison of AMPS with behavioural clones using different generalisation methods in Corner World. Shown are the expected number of problems solved in 100 episodes without a teacher.

static decision graphs in this way does not lead to good policies. Decision graph behavioural clones only solve 36.62 ± 5.55 on average of the first 100 episodes following three training episodes, whereas AMPS with decision graph partitioning solves 82.45 ± 2.65 on average.

In the Corner World domain, a behavioural clone may learn using decision graph or nearest neighbour generalisation. If the agent induces a decision graph, the decision graph can be extremely large and unnecessarily complex. In the experiments, the decision graph after two training episodes carves the state space into approximately 250 regions. The performance of decision graph clones is very poor in comparison to AMPS. If the behavioural clone uses nearest neighbour generalisation with a Euclidean distance metric, the performance is still quite poor. Table 10.1 compares the performance of AMPS and behavioural clones with decision graph and nearest neighbour generalisation.

Agent	Perf.
Random	0.30±0.12
Q(λ)	0.27±0.12
Sarsa(λ)	0.27±0.12
Abstract Q(λ)	45.93±3.50
Abstract Sarsa(λ)	43.83±3.67
AMPS	82.45±2.65

Table 10.2: A comparison of the performance of several temporal-difference learning algorithms against AMPS in the Taxi World domain. Shown are the expected number of problems the various kinds of agents can solve in 100 episodes. In the experiments λ is set to 0.8.

10.4.2 Temporal-Difference Learning

When the state or action space is large, tabular temporal-difference learning (see Section 3.4.3) is not likely to perform well. The success of tabular temporal-difference learning depends on the agent trying every action from every state multiple times. In the Taxi World and Corner World domains, tabular temporal-difference learning results in behaviour not much better than random.

As Section 3.4.3 mentions, one may combine temporal-difference learning with a Monte Carlo approach using eligibility traces. Several algorithms such as Q(λ) due to Watkins (1989) and Sarsa(λ) due to Rummery (1995) use eligibility traces to improve performance.[2] Even with eligibility traces, performance can be quite poor without generalisation.

A domain expert may manually discretise large or continuous state and action spaces and then perform tabular temporal-difference learning over the abstract states and actions. Table 10.2 summarises some results in the Taxi World domain using Q(λ) and Sarsa(λ) with a static abstraction.[3] The table reveals that using the abstraction greatly improves performance. However, neither Q(λ) nor Sarsa(λ) with abstraction performs nearly as well as AMPS. AMPS has the advantage because it uses a dynamic abstraction and explicitly learns a model of the system dynamics over which it may plan.

[2]For an introduction to temporal-difference learning with eligibility traces, see the text by Sutton and Barto (1998) along with the errata available at `http://www.cs.ualberta.ca/~sutton/book/errata.html`.

[3]The discretisation in these experiments is the finest partition allowed by the separation mechanism used by AMPS in this domain.

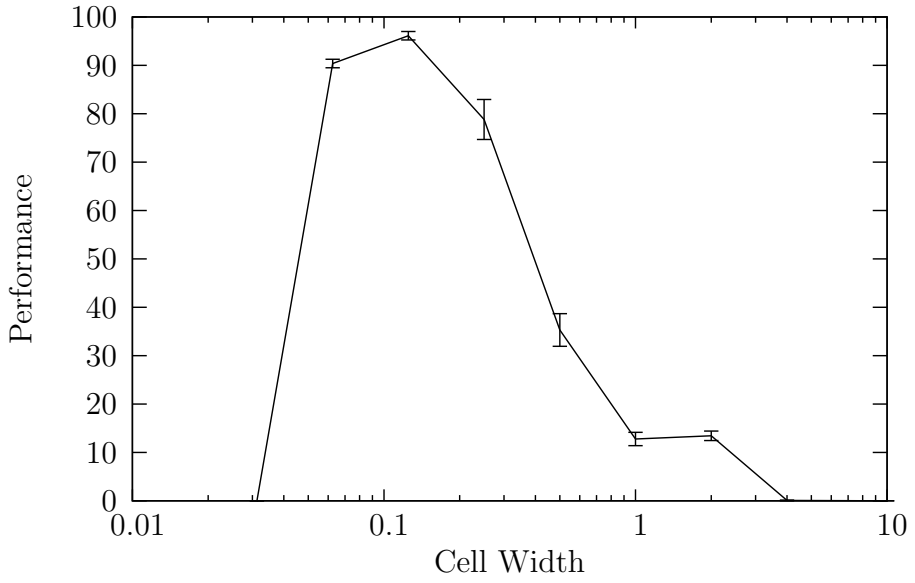


Figure 10.20: A plot indicating how the coarseness of the discretisation in the Corner World problem affects the performance of Q(λ). In these experiments, the 10×10 m track was discretised at different resolutions. The horizontal axis represents different cell widths in metres. Performance is measured according to the average number of successes in 100 episodes. Again, λ is set to 0.8.

Another set of experiments investigate the performance of temporal-difference learning with eligibility traces in the Corner World domain. Since the state space is continuous and the sample rate of the agent is finite, the agent will almost never find itself in a previously experienced state. Hence, temporal-difference learning without generalisation will perform no better than an agent that moves randomly throughout the state space.

With an abstraction at the right level of detail, temporal-difference learning can learn competent behaviour remarkably quickly. The experiments apply Q(λ) to a grid-based discretisation of Corner World with variable cell widths. Unlike the earlier experiments with AMPS in Corner World, the agent is provided with only four actions corresponding to moving in each of the cardinal directions. Restricting the available actions greatly simplifies the problem. Figure 10.20 plots the cell width in the discretisation against performance. Clearly, the level of abstraction greatly affects performance.

The success of Q(λ) at the right level of abstraction in Corner World is due to the use of an eligibility trace and teacher as well as the structure of the problem.

The first two training episodes result in value updates in the cells along the trajectories of the teacher. These value updates usually result in reasonable policy actions at the visited cells. When the agent is left to use Q(λ) on its own, it will flail around until it reaches a cell visited by the teacher. If the agent chooses an on-policy action, it will generally head in the right direction. If the agent loses track of the trail left by the teacher, random exploration will bring it back on track, so long as the discretisation is sufficiently coarse. If the discretisation is too coarse the agent cannot make the necessary perceptual distinctions to navigate around the corner.

One of the main challenges of applying tabular temporal-difference learning is determining the right level of abstraction. The advantage of AMPS is that it learns the appropriate level of abstraction while interacting with the environment.

As Chapter 4 discusses, temporal-difference learning can be used with generalisation approaches other than abstraction. Goebel (2005) investigates the use of self-organising maps (see Section 4.3) and neural networks (see Section 4.4) as generalisation methods for temporal-difference learning. His thesis shows that these approaches perform rather poorly on both the Corner World and Taxi World tasks.

10.4.3 Prioritised Sweeping

Unlike temporal-difference learning, prioritised sweeping (Moore and Atkeson, 1993) is a model-based reinforcement learning algorithm. As Section 2.6 discusses, model-based approaches generally learn better policies with less experience than model-free approaches. Model-free approaches, such as those considered in Section 10.4.2, only update the value function along experienced trajectories, whereas model-based approaches can perform updates whenever the estimated model indicates it is necessary.

Figure 10.21 represents a portion of a problem that illustrates the advantage of a model-based approach. Suppose that s_5 is a terminal state with positive reward and that s_3 is a terminal state with negative reward. If the agent experiences the sequence

$$s_1, s_2, s_3, s_4, s_2, s_5,$$

a model-free approach would not be able to update the value of s_1 after experiencing the positive reward in s_5 because of the termination in s_3. Model-based

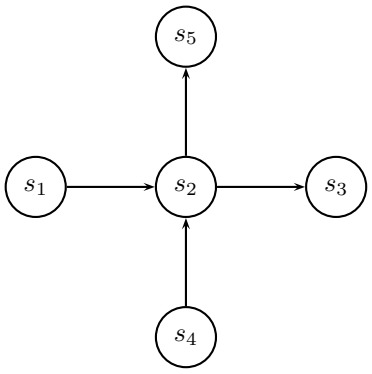


Figure 10.21: A problem illustrating the advantage of a model-based approach.

methods, on the other hand, may update the value of s_1 after receiving reward in s_5. With a better estimate of the value function from the limited experience, model-based algorithms are expected to produce better behaviour. The literature contains many other examples of model-based approaches outperforming model-free approaches (e.g., Moore and Atkeson, 1993; Peng and Williams, 1993; Atkeson and Santamaría, 1997).

Implementing prioritised sweeping based on an existing implementation of AMPS is relatively easy. Prioritised sweeping is essentially AMPS with a static map that partitions the state and action spaces. Since the partitions do not change, there is no reason to retain old experiences in memory after updating the model estimate. AMPS, in contrast, retains as much experience in memory as possible so that it can use the potentially costly experience to update the model following map revision (Section 9.4.1). If AMPS did not retain its experience in memory, it would have to relearn the model in areas altered by the map-revision process.

Although prioritised sweeping typically consumes much less memory than AMPS, its primary limitation is that it requires a domain expert to provide a discretisation of the state and action spaces at a suitable resolution. AMPS, in contrast, attempts to learn the correct discretisation from experience. For some problems, it is easy to determine a suitable discretisation, but for other problems, it is not so straightforward. It is important to use a suitable discretisation.

AMPS, because of its use of dynamic abstraction, is able to outperform prioritised sweeping on the Taxi World task. Without an abstraction, just as with the temporal difference algorithms, prioritised sweeping performs randomly. With

the same abstraction Section 10.4.2 uses with the Taxi World experiments, prioritised sweeping is expected to solve 67.96 ± 3.30 problems, which is about 50% more than $Q(\lambda)$ and Sarsa(λ) with abstraction. AMPS, which learns its own abstractions, can solve on average 82.45 ± 2.65 problems.

Prioritised sweeping can learn competent behaviour very quickly for the Corner World task with an appropriate level of abstraction. As with the experiments in Section 10.4.2 involving Q(λ), the experiments with prioritised sweeping use a grid-based discretisation of the state space and an alternative version of the problem with only four actions. As can be expected, prioritised sweeping performs better than temporal-difference learning with eligibility traces over a wider range of resolutions. For Q(λ) to perform with 90% competency the cells have to be between 1/16 and 1/8 m wide, but for prioritised sweeping to perform with 90% competency the cells may be between 1/16 to 2 m wide.

10.4.4 UTree

UTree (McCallum, 1995, 1996a) was among the first model-based abstraction algorithms to appear in the literature and has served as an inspiration in the development of AMPS. Section 4.2 provides an overview of UTree, and this section examines the algorithm in greater detail and explains how it relates to AMPS.

The Java Reinforcement Learning Framework (JRLF), which was developed as part of this thesis (see Section 9.2), includes an implementation of the basic ideas underlying UTree. The version of UTree in JRLF is not intended to exactly replicate the work of McCallum, and there are a few significant differences:

- **Representation:** McCallum assumes an attribute-value representation. The JRLF version of UTree borrows much of its implementation from AMPS, allowing it to use more general representations such as real-valued vectors as in the Continuous U-Tree algorithm (Uther and Veloso, 1998; Uther, 2002).

- **Modelling and planning:** McCallum uses MDPs to model the system dynamics, but the JRLF version of UTree uses SMDPs. An SMDP is a continuous-time generalisation of an MDP (Section 3.2) and is suitable for a wider variety of problems. McCallum uses a single sweep of value iteration to update the value function and policy, but the JRLF version of UTree uses the same prioritised value iteration algorithm as AMPS.

- **Historical distinctions:** McCallum allows for the introduction of historical distinctions. Instead of only splitting on attributes of the current state, his version of UTree can split on attributes of previously observed states. The current version of UTree as part of JRLF does not include this functionality, but it is an area of further research (Section 11.3.2). The problems in this dissertation do not require the use of historical distinctions to learn a competent policy; all of the relevant information for making a good decision is contained in the current state.

- **Significance tests:** McCallum only uses the Kolmogorov-Smirnov statistical test when deciding when and how to split a region. The version of UTree packaged with JRLF can use a variety of different statistical tests, but uses sum-squared error by default like the Continuous U-Tree algorithm (Uther and Veloso, 1998; Uther, 2002).

- **Prioritisation:** McCallum introduces splits to any region where the split is deemed significant above a certain threshold by the Kolmogorov-Smirnov test statistic. The JRLF version of UTree prioritises its introduction of perceptual distinctions.

Figure 10.22 compares UTree against AMPS on ten episodes of the Taxi World problem following three training episodes. To control for the use of failure sensing by AMPS, the performance of a version of UTree with failure revision is also shown in the figure as UTree+F. Both versions of UTree do not perform as well as AMPS[4] and require much more time to run, as Figure 10.23 shows.

There are several reasons why AMPS performs better than UTree with respect to both behavioural competency and real-time computation.

- **Model simplification:** Unlike UTree, AMPS attempts to simplify its model when experience indicates it might be useful (Section 7.2). As the experiments in Section 10.3.2 indicate, simplification can greatly improve performance. Not only do simpler models require less memory to represent and less time to compute their optimal policies, but they are able to better generalise from limited experience. UTree, like other abstraction methods for reinforcement learning, does not attempt to simplify its model in the same way as AMPS.

[4]The p-values for the Wilcoxon signed-rank test (Wilcoxon, 1945) when comparing AMPS with UTree and Utree+F are 3.8×10^{-6} and 0.038 respectively.

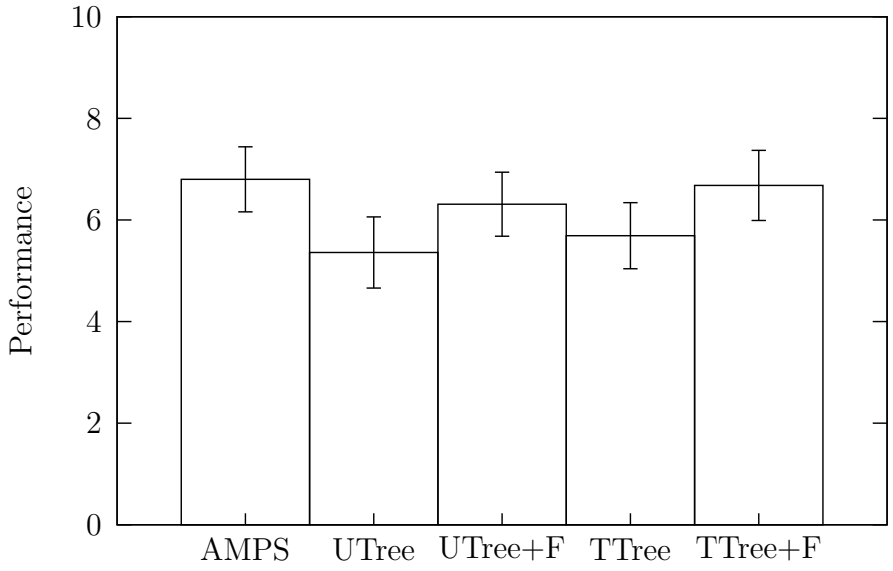


Figure 10.22: A comparison of the performance of various algorithms on the Taxi World problem. Performance is measured by the expected number of problems solved in ten episodes following three episodes of training.

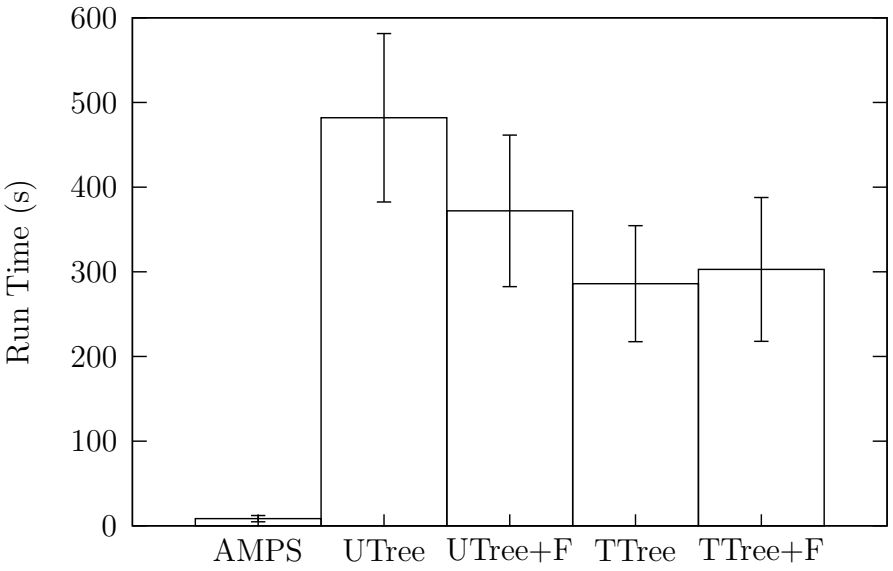


Figure 10.23: A comparison of expected run times of various algorithms on the Taxi World problem.

- **Model-based heuristics:** AMPS uses model-based splitting heuristics, but UTree uses a samples-based splitting heuristic. To decide when to split, AMPS only needs to inspect the estimated model, which can be done very quickly. If AMPS decides to introduce a perceptual distinction, it uses an approach based on supervised learning (Chapter 6), which is typically very fast. In contrast with the model-based heuristics of AMPS, the samples-based heuristic that UTree uses can be computationally expensive. Before UTree can determine whether to split a particular region, it must estimate the value of each individual sample within the region. UTree estimates the value of each sample according to Equation 4.1. Unfortunately, it is not possible to cache these estimated values with the individual observations because they depend on the global structure of the model and the current plan. Once UTree computes the values of all of the samples in the region, it iterates through every possible way of splitting the samples. For each way of splitting the samples into two or more new regions, UTree computes the difference between the resulting distributions of sample values. If these distributions are significantly different, UTree will introduce the split.

- **Trajectory heuristics:** UTree estimates the value of a sample based on the immediate reward associated with the sample and the value of the region to which the next sample in the trajectory belongs. If the agent only receives non-zero reward at the goal, as is the case in the Taxi World and Corner World domains, then the only difference in sample value will be at the edges of regions. Hence, as Uther (2002, Section 3.5) observes, the splits that UTree introduces only occur at the edges of regions, resulting in the creation of many more regions than necessary. AMPS avoids this problem in a manner similar to TTree by introducing distinctions based on trajectories instead of individual samples.

10.4.5 TTree

TTree (Uther, 2002; Uther and Veloso, 2003) is much like UTree, but it introduces splits based on information from entire trajectories instead of individual observations. The original purpose of TTree was to compute optimal policies for complex SMDPs by leveraging a small collection of supplied policies. TTree was not intended as an algorithm for learning good behaviour through interaction. In-

stead, it assumes access to a generative model of the world. This section adapts the basic ideas of the original TTree algorithm for use in interactive problems where the dynamics are initially unknown.

JRLF contains an implementation of TTree for the purpose of comparison with AMPS. As with the JRLF version of UTree, the JRLF version of TTree shares much of its implementation with AMPS. Besides the lack of model simplification in TTree, the main difference between TTree and AMPS is the way they introduce splits. The JRLF version of TTree introduces splits based on the estimated value of samples, just like UTree. However, the estimation of sample values is different between TTree and UTree.

TTree computes the value of a sample state s in region S in the following way. Let R be the accumulated discounted reward the agent receives while transitioning from s in S to some other region, and let V be the estimated value of the resulting region. If β is the continuous-time discount rate and t is the amount of time required to transition from s in S to another region, then the value of the sampled state s is $R + e^{-\beta t}V$.

As with UTree, the TTree algorithm may use any heuristic measure of significance to decide when and how to introduce a perceptual distinction. By default, JRLF uses sum-squared error as it does with UTree. The original version of TTree, however, uses a minimum message length heuristic.

As Figure 10.22 shows, TTree performs marginally better than UTree ($p = 0.08$), but not as well as AMPS ($p = 2.47 \times 10^{-5}$). When the agent uses TTree with failure revision, it can perform almost as well as AMPS on this problem, ignoring issues with computation. As Figure 10.23 reveals, the JRLF implementation of TTree requires around 35 times the computation of AMPS. The inefficiency of TTree is due to the same reasons as UTree (Section 10.4.4).

10.5 Discussion

The first part of this chapter demonstrates that AMPS is a flexible framework that can learn quickly in an environment with little guidance from an imperfect teacher. The second part of this chapter examines the contributions of the various components in AMPS. The third part of this chapter compares AMPS with other methods. This section summarises and discusses the results of this chapter.

This chapter shows AMPS working on different kinds of problems. The Goal

World domain has a continuous state space and a discrete action space, the Taxi World domain has a large discrete state space and a small discrete action space, and the Corner World has both a continuous state space and a continuous action space. The same basic implementation of AMPS can solve all of these problems with only the map structure (Section 5.2) requiring changing between problems. It is the map structure that generalises from the underlying representation of the state and action spaces. In the Goal World and Taxi World problems, the map is represented by a decision graph with tests generated automatically from an XML file that specifies a set of typed functions and relations. This chapter shows AMPS using both nearest neighbour and decision graph generalisation approaches for the Corner World domain.

AMPS learns competent policies for all three kinds of problems. It learns a reasonable policy for the Goal World problem almost immediately from a single training episode. Learning good policies in the more difficult Taxi World and Corner World problems requires much more effort on the part of AMPS. During the early stages of learning in these domains, the behaviour of AMPS is far from optimal because its model is too simplistic. However, AMPS is able to quickly use its experience to adapt its model and revise its plan. When AMPS exceeds its specified memory capacity, it removes old experiences, which can lead to simplifications in the model. When the agent encounters novel experiences that it cannot explain well with its existing model, AMPS increases the complexity of the model to account for the new experience by using its splitting heuristics (Section 6.2).

The two kinds of splitting heuristics, value revision and failure revision, work together in AMPS to introduce the appropriate perceptual and actional distinctions. For some problems, such as the Taxi World problem, the combined use of both heuristics is necessary for competent behaviour. Other problems rely primarily on one of the heuristics. For example, the Corner World problem introduces most of its distinctions using failure revision.

Model simplification is an important idea in AMPS as demonstrated in the experiments in this chapter. Without simplification, the model can grow very quickly. Simpler models require less memory and less computation and provide better generalisation from limited experience. AMPS also includes routines for simplifying the map data structure when represented by decision graphs (Section 7.3). The experiments in this chapter show that decision graph simplification

algorithms can significantly reduce the number of nodes in the graph, speeding classification and reducing memory requirements.

The experiments in this chapter show that value clipping, a novel algorithm introduced in this thesis (Section 6.6), can greatly improve performance. This improvement in performance is due to the ability of the algorithm to identify when there does not exist a path in the model from the current region to the goal region, indicating the agent must rely upon random exploration. The experiments also show that increasing the number of planning updates per step does not typically improve performance, supporting the observations made at the beginning of Chapter 8 with static abstractions.

This chapter compares AMPS to several other approaches. The experiments show that behavioural clones are not robust to imperfect teachers. Since behavioural clones are unable to adapt to new situations on their own, they can hardly be considered intelligent using the definition introduced in Chapter 1. Temporal-difference learning and prioritised sweeping both require abstractions to be provided. If these methods have abstractions at the right level of detail, they can perform quite well. Temporal-difference learning has been used with function-approximation methods elsewhere (Goebel, 2005) and has been shown to have difficulty on both the Taxi World and Corner World problems.

Of the algorithms in the literature, UTree and TTree, as adapted in JRLF, most closely resemble AMPS in that they adaptively partition the state space and use model-based planning. AMPS, as has been discussed, is much more general than both of these approaches and is capable of generalising in both the state and action spaces and both splitting and merging existing regions. The way in which UTree and TTree introduce perceptual distinctions is fundamentally different from AMPS. The experiments show that AMPS is much faster than both UTree and TTree and can learn competent behaviour from less experience.

Depending on the structure of the task, what is known about the problem, and the computational resources available to the agent, AMPS may not be the best approach. For example, if a good discretisation of the state and action spaces is known, then prioritised sweeping by itself may perform better than AMPS. If the cost of experience is negligible, then one of the other generalisation approaches based on temporal-difference learning (Sections 4.3 and 4.4) can do just as well. If a generative model of the problem is available, then another approach may be appropriate (Ng and Jordan, 2000). AMPS is best suited for problems where

experience is expensive and the agent possesses little prior knowledge and modest computational resources.

Chapter 11

Summary and Further Work

Before one can truly understand intelligence, one must be able to create it. As written on Feynman's blackboard at the time of his death, "What I cannot create, I do not understand."[1] The approach suggested in this thesis is a step towards understanding the computational processes involved in learning intelligent behaviour. This chapter summarises the approach taken in this thesis, highlights the contributions made, and outlines several areas of further research.

11.1 Summary

An intelligent agent must be able to use its past experience to develop an understanding of how its actions affect the world in which it is situated. Given some objective, the agent must be able to effectively use its understanding of the world to produce a plan that is robust to the uncertainty present in the world. This section briefly summarises the basic ideas underlying AMPS that aim to meet these requirements for intelligence.

There are many different ways an agent may represent its "understanding of the world." A particularly powerful and well-studied way to model the behaviour of the world is with a continuous-time, discrete-event model known as a semi-Markov decision process (SMDP). SMDPs are suitable for modelling environments with stochastic state transitions, variable transition durations, and reward. One may use dynamic programming to efficiently compute optimal policies.

[1] A photograph of Feynman's blackboard including this quote is contained in *The Universe in a Nutshell* (Hawking, 2001, page 83).

The challenge of the agent is to use its experience in the world to estimate an SMDP. In problems with large state and action spaces, the agent must generalise from its limited experience by grouping together similar states and actions, effectively partitioning the state and action spaces into finite sets of regions. This process is called abstraction, and several different abstraction approaches have been proposed in the literature.

The existing abstraction approaches have many limitations. They generally only increase resolution, require a large amount of data before changing the abstraction, only generalise over states, and are computationally expensive. AMPS aims to solve these problems using a new kind of approach.

AMPS splits and merges existing regions in its abstraction according to a set of heuristics. If the system detects a significant difference in value along different trajectories associated with a single state-action region, it will attempt to split the state or action space using a mechanism related to supervised learning. This mechanism is defined generally, allowing AMPS to leverage a wide variety of representations. The system also splits regions according to observed failure and merges regions by inspecting the transitions between regions and the current plan.

At any given time, there may be many different regions requiring revision. AMPS prioritises these revisions so that as soon as the agent has computational resources available, it will perform the appropriate updates. Changes in the abstraction lead to changes in the model, requiring changes to the plan. AMPS prioritises the planning process, and when the agent has time, AMPS performs dynamic programming updates in high-priority regions.

AMPS is a general and flexible system for integrating modelling and planning for learning adaptive behaviour. The system efficiently revises its model, efficiently revises its plan, and efficiently generalises from limited experience. One may apply AMPS to a wide variety of problems without having to reimplement the core system when changing representations.

11.2 Contributions

The first part of this thesis provides a broad introduction to various approaches for building agents (Chapter 2), systems for modelling dynamic environments (Chapter 3), and methods for generalisation in reinforcement learning problems (Chapter 4). These background chapters do not advance any new ideas, but

they do present an original synthesis of existing work from different research communities.

The remainder of this thesis makes the following contributions:

- **A novel approach that connects reinforcement learning to supervised and unsupervised learning techniques for the purpose of generalisation.** Although other reinforcement learning algorithms have used unsupervised learning techniques (e.g., self-organising maps, Section 4.3.2) and data structures frequently used in supervised learning (e.g., neural networks, Section 4.4), the way in which this thesis combines supervised and unsupervised learning with reinforcement learning is original (see Sections 5.2, 6.2, and 6.3).

- **A flexible framework for leveraging different kinds of representations.** Generalisation from limited experience depends upon being able to extract structure from the underlying representation. Chapter 6 discusses ways of tailoring AMPS to the representation using decision graph and nearest neighbour approaches, and Chapter 10 shows how these approaches can be used in practice.

- **An efficient model-based implementation that learns competent behaviour from little experience.** Chapter 10 shows that AMPS can learn extremely quickly on different kinds of problems. AMPS appears to be the first system that prioritises both planning and modelling. Existing model-based algorithms such as UTree (Section 4.2.4) and TTree (Section 4.2.6) are not nearly as efficient or flexible as AMPS as discussed at the end of Chapter 10.

- **An abstraction system that dynamically increases and decreases resolution in both the state space and action space.** The abstraction methods surveyed in Section 4.2 only introduce new perceptual distinctions as the agent accumulates experience; they never decrease resolution when further experience indicates doing so might be useful. AMPS appears to be the first abstraction approach to perform generalisation in the action space, although some local and parametric approximation methods do attempt action space generalisation (see Sections 4.3 and 4.4).

- **Supporting algorithms for efficient modelling and planning while adapting abstractions.** Section 5.3 presents different ways of estimating the model from experience, and Chapter 8 discusses several prioritised value iteration algorithms that AMPS may use to revise its plan according to its changing model. This thesis also introduces value clipping (Section 6.6), algorithms for simplifying models (Section 7.2), and algorithms for simplifying decision graphs (Section 7.3).

- **An implemented system for testing, visualising, and comparing the approach to others.** The implementation of AMPS is built on top of the Java Reinforcement Learning Framework (JRLF) as described in Section 9.2. Although JRLF was developed specifically to test the hypotheses of this thesis, the framework is likely to be helpful to other researchers in testing and comparing their reinforcement learning algorithms and extending AMPS.

11.3 Further Work

Since AMPS is a new approach and has been tested on relatively few problems, further research is necessary to determine how well the approach scales to more complex problems. Besides applying the existing implementation of AMPS to new problems, there are several other promising lines of further research.

11.3.1 Logical Decision Trees

As Section 4.2.8 discusses, relational reinforcement learning algorithms partition the state space using logical decision trees. The decision nodes in a logical decision tree are allowed to contain logical tests that refer to variables introduced higher in the tree. For some kinds of problems, the extra expressive power of logical decision trees can allow them to represent some kinds of partitions more compactly than regular decision trees.

AMPS currently only uses standard decision graphs. It might be interesting to see how the generalisation of logical decision trees to logical decision graphs will fare on problems with relational structure such as Blocks World. In addition to using logical decision graphs to partition the state space, it would be interesting to use logical decision graphs to partition the action space. One could allow the

graph that partitions the action space to refer to variables introduced by the graph that partitions the state space.

11.3.2 Historical Distinctions

Currently, AMPS only makes distinctions based on the current observation. Depending on the problem, it might be useful to make distinctions based on past observations or actions, especially when the environment is only partially observable (Section 3.5). The literature contains model-based approaches (McCallum, 1995, 1996a; Au, 2005) and model-free approaches (Lin and Mitchell, 1992; McCallum, 1996b; Wiering and Schmidhuber, 1997) that make historical distinctions. The literature also contains examples of partially observable problems where memoryless policies still produce satisfactory behaviour (Littman, 1994; Loch and Singh, 1998). Nevertheless, it would be interesting to investigate how one might extend AMPS to make historical distinctions.

Previous algorithms that make historical distinctions assume that the state of the world evolves in discrete stages. Distinctions are typically composed of a test and a history index, indicating how many stages backwards from the current stage the test should be applied. However, AMPS does not assume that the world evolves in stages. Instead, AMPS assumes that the world evolves continuously and that the state is sampled at some finite frequency.

If AMPS is to make historical distinctions with decision graphs, it is necessary to decide upon a temporal logic to represent these distinctions. For example, one might introduce the variable t_0 to represent the current time and the predicate $holds(\phi, t)$ to represent the fact that the test ϕ holds at time t. A historical distinction might be represented by the sentence

$$\exists t. \left[(t > t_0 - 1) \wedge (t < t_0) \wedge holds(\phi, t)\right],$$

which means that ϕ holds at some point during the last unit of time. One can imagine building quite an expressive language, such as the one defined by Allen (1984), but the challenge is to make it so that AMPS can efficiently choose which historical distinction to introduce. Even a relatively simple temporal logic risks being computationally expensive and prone to overfitting.

AMPS with nearest neighbour generalisation might also be extended to make historical distinctions using ideas from the nearest sequence memory algorithm (McCallum, 1996b). The nearest sequence memory algorithm measures similarity

between *observation chains* instead of just observations. Although the distance metric suggested by McCallum is likely to be useful only in environments with small state spaces with little noise, it might be possible to enhance the distance metric so that it works with AMPS.

11.3.3 Experience Consolidation

The experience-revision process (Section 9.4.2) removes old experiences from memory to keep from exceeding some specified memory limit and to reduce computational demands. It would be interesting to investigate ways of consolidating old experiences instead of disposing of them completely. For example, if a region of the state space stabilises (i.e., the revision process has not recently split or merged the region), it might be useful to purge the experiences associated with the region and symbolically encode transition probabilities and other model parameters. Instead of keeping fifty trajectories from one region to another in memory, the consolidation process could simply encode the fact.

Much work is necessary to determine when and how to perform experience consolidation in AMPS, and it is likely to be a promising area of further research. There is some evidence from neurobiology that humans perform some form of consolidation. Over time, internal representations of the world in short-term memory become encoded as molecular or structural modifications in the brain that persist as long-term memories (Kandel, 2001; Lee et al., 2004). Although this thesis is primarily concerned with solving the problem of intelligence through traditional computational means, exploring how the processes in AMPS compare with human cognition and neurophysiology may be fruitful.

11.3.4 Parallelisation

AMPS, like most software today, is designed specifically for a single processor that executes a stream of sequential instructions that have access to a single bank of memory. For several decades, advances in microprocessor design have led to exponential increases in performance. Making processors faster involves either increasing the clock rate or exploiting instruction-level parallelism through pipelining or superscalar architectures (Patterson and Hennessy, 2005). Unfortunately, it is becoming increasingly difficult for microprocessor designers to continue to exploit these techniques for several reasons outlined by Olukotun and

Hammond (2005). Hence, the future of computing will involve multiprocessor architectures.

The modelling and planning processes in AMPS may be executed in parallel, across multiple processors. Different state regions may be distributed across processors. Instead of maintaining a single priority queue of updates over the entire state space, each processor would maintain its own priority queue and perform updates on its own collection of state regions. Plan updates may be performed in parallel (Bertsekas and Tsitsiklis, 1997; Wingate and Seppi, 2004). Further research would address various parallelisation issues such as how to best distribute state regions across processors and how to minimise communication between processors to enhance scalability.

Appendix A

Proofs

Theorem 1 (Contraction) *For any two value functions V and V' and any policy π,*

$$\|BV - BV'\| \leq \alpha \|V - V'\| \tag{A.1}$$

$$\|B_\pi V - B_\pi V'\| \leq \alpha \|V - V'\|. \tag{A.2}$$

Proof Equation A.1 is shown easily using the definition of α in Equation 3.5.

$$\begin{aligned} BV(s) &= \max_{a \in \mathbb{A}(s)} \left[R(s,a) + \sum_{s' \in \mathbb{S}} P(s' \mid s,a)\gamma(s,a,s')V(s') \right] \\ &= \max_{a \in \mathbb{A}(s)} \left[R(s,a) + \sum_{s' \in \mathbb{S}} P(s' \mid s,a)\gamma(s,a,s') \left[V(s') - V'(s') + V'(s')\right] \right] \\ &= BV'(s) + \max_{a \in \mathbb{A}(s)} \sum_{s' \in \mathbb{S}} P(s' \mid s,a)\gamma(s,a,s') \left[V(s') - V'(s')\right] \\ &\leq BV'(s) + \alpha \|V - V'\| \end{aligned}$$

Therefore, $\|BV - BV'\| \leq \alpha \|V - V'\|$. The proof for Equation A.2 is similar. ■

Theorem 2 *Given any value function V and the mapping B,*

$$\lim_{t \to \infty} B^t V = V^* \tag{A.3}$$

$$\lim_{t \to \infty} B_\pi^t V = V^\pi \tag{A.4}$$

and

$$BV^* = V^* \tag{A.5}$$

$$B_\pi V^\pi = V^\pi. \tag{A.6}$$

Proof Since B is a contraction mapping, it follows from the Contraction Mapping Theorem (Luenberger, 1969, pages 272–273) that $B^t V$ converges to a unique fixed point $\tilde{V}$ and that $B\tilde{V} = \tilde{V}$. To prove Equations A.3 and A.5, it is necessary to show $\tilde{V} = V^*$. The following argument is similar to that of Bertsekas and Tsitsiklis (1997, pages 316–317) for MDPs. Let V_0 be a value function that maps all states to 0. For any policy π, it follows from the definition of B_π and B that $B_\pi V_0 \leq B V_0$. Likewise, $B_\pi^k V_0 \leq B^k V_0$. Because $B_{\pi^*}^k V_0 \to V^*$ and $B^k V_0 \to \tilde{V}$ as $k \to \infty$, it follows that $V^* \leq \tilde{V}$.

With a policy π chosen such that $B_\pi \tilde{V} = B\tilde{V}$, it follows from $B\tilde{V} = \tilde{V}$ that $B_\pi^k \tilde{V} = \tilde{V}$. Since $B_\pi^k \tilde{V} \to V^\pi$ as $k \to \infty$, it follows that $V^\pi = \tilde{V}$, which along with $V^\pi \leq V^*$ shows that $\tilde{V} \leq V^*$. Because $V^* \leq \tilde{V}$ and $\tilde{V} \leq V^*$, it must be that $V^* = \tilde{V}$.

Equations A.4 and A.6 are proven in a similar manner. ■

Theorem 3 *If the policy π is defined such that*

$$\pi(s) = \arg\max_{a \in \mathbb{A}(s)} \left[R(s,a) + \sum_{s' \in \mathbb{S}} P(s' \mid s, a) \gamma(s,a,s') V^*(s') \right], \tag{A.7}$$

then π is optimal.

Proof Suppose Equation A.7 holds. It follows that $B_\pi V^* = B V^*$, and by Equation A.5 of Theorem 2, $B_\pi V^* = V^*$. Likewise, $B_\pi^k V^* = V^*$. As $k \to \infty$, $B_\pi^k V^* \to V^\pi$ by Equation A.4. Hence, $V^\pi = V^*$ and therefore π is optimal. ■

Theorem 4 *If $V = BV'$ and $\|V - V'\| \leq \epsilon(1-\alpha)/(2\alpha^2)$ then $\|V^* - V^\pi\| \leq \epsilon$, where π is the greedy policy calculated from V.*

Proof The arguments in this proof are similar to those of Williams and Baird (1993). This proof assumes $\|V - V'\| \leq \epsilon(1-\alpha)/(2\alpha^2)$, and then shows that $\|V^* - V^\pi\| \leq \epsilon$. Here, π is a greedy policy calculated from V, and π^* is an optimal policy as calculated in Equation A.7.

By the triangle inequality, Theorem 1, and Theorem 2,

$$\begin{aligned}
\|V^* - V\| &\leq \|V^* - BV\| + \|BV - V\| \\
&\leq \alpha \|V^* - V\| + \alpha \|V - V'\| \\
&\leq \alpha \|V^* - V\| + \epsilon(1-\alpha)/(2\alpha) \\
&\leq \epsilon/(2\alpha).
\end{aligned}$$

Consequently,

$$V^*(s) \leq V(s) + \epsilon/(2\alpha). \tag{A.8}$$

Similarly,

$$\begin{aligned}
\|V - V^\pi\| &\leq \|V - B_\pi V\| + \|B_\pi V - V^\pi\| \\
&\leq \|V - B_\pi V\| + \alpha \|V - V^\pi\| \\
&\leq \|V - B_\pi V\| / (1-\alpha).
\end{aligned}$$

Since π is greedy for V, it follows that $B_\pi V = BV$ and therefore

$$V(s) \leq V^\pi(s) + \epsilon/(2\alpha). \tag{A.9}$$

Since π is greedy for V,

$$\begin{aligned}
& R(s, \pi^*(s)) + \sum_{s' \in \mathbb{S}} P(s' \mid s, \pi^*(s)) \gamma(s, \pi^*(s), s') V(s') \\
\leq\ & R(s, \pi(s)) + \sum_{s' \in \mathbb{S}} P(s' \mid s, \pi(s)) \gamma(s, \pi(s), s') V(s').
\end{aligned} \tag{A.10}$$

The equalities of Equations A.8, A.10, and A.9 produce:

$$\begin{aligned}
V^*(s) &= R(s, \pi^*(s)) + \sum_{s' \in \mathbb{S}} P(s' \mid s, \pi^*(s)) \gamma(s, \pi^*(s), s') V^*(s') \\
&\leq R(s, \pi^*(s)) + \sum_{s' \in \mathbb{S}} P(s' \mid s, \pi^*(s)) \gamma(s, \pi^*(s), s') \left[V(s') + \epsilon/(2\alpha)\right] \\
&\leq \epsilon/2 + R(s, \pi^*(s)) + \sum_{s' \in \mathbb{S}} P(s' \mid s, \pi^*(s)) \gamma(s, \pi^*(s), s') V(s') \\
&\leq \epsilon/2 + R(s, \pi(s)) + \sum_{s' \in \mathbb{S}} P(s' \mid s, \pi(s)) \gamma(s, \pi(s), s') V(s') \\
&\leq \epsilon/2 + R(s, \pi(s)) + \sum_{s' \in \mathbb{S}} P(s' \mid s, \pi(s)) \gamma(s, \pi(s), s') \left[V^\pi(s') + \epsilon/(2\alpha)\right] \\
&\leq \epsilon + R(s, \pi(s)) + \sum_{s' \in \mathbb{S}} P(s' \mid s, \pi(s)) \gamma(s, \pi(s), s') V^\pi(s') \\
&= \epsilon + V^\pi(s).
\end{aligned}$$

Therefore $\|V^* - V^\pi\| \leq \epsilon$. ■

Theorem 5 *When Algorithm 8 terminates, the expected discounted return when following the greedy policy calculated from the resulting value function will be within ϵ of optimal.*

Proof Let $s_1, \ldots, s_n$ be a sequence of states in the order their value function was updated during the last iteration before termination. Let $V_1, \ldots, V_n$ be the sequence of value functions in the order they were updated during the last iteration before termination. Set $V_0 = V'$ to be the value function at the beginning of the final iteration, and set $V_n = V$ to be the value function at the end of the final iteration. The updates follow

$$V_k(s_h) = \begin{cases} BV_{k-1}(s_h) & \text{if } k = h \\ V_{k-1}(s_h) & \text{otherwise} \end{cases} . \tag{A.11}$$

This proof proceeds by assuming that the stopping criterion has been met and then showing that $\|V^* - V^\pi\| \leq \epsilon$, where π is the greedy policy calculated from V. If the stopping criterion has been met, then for all k,

$$|V_k(s_k) - V_{k+1}(s_k)| \leq \epsilon(1-\alpha)/(2\alpha^2).$$

The triangle inequality, Equation A.11, Theorem 1, and Theorem 2, show that

$$\begin{aligned} |V_k(s_k) - V^*(s_k)| &\leq |BV_k(s_k) - V_k(s_k)| + |BV_k(s_k) - V^*(s_k)| \\ &\leq \alpha\,|V_k(s_k) - V_{k-1}(s_k)| + \alpha\,|V_k(s_k) - V^*(s_k)| \\ &\leq \epsilon(1-\alpha)/(2\alpha) + \alpha\,|V_k(s_k) - V^*(s_k)|\,. \end{aligned}$$

Hence, $V^*(s_k) \leq V_k(s_k) + \epsilon/(2\alpha)$ for all k. Since $V_k(s_k) = V(s_k)$, it follows that $V^*(s) \leq V(s) + \epsilon/(2\alpha)$. The remainder of the proof follows that of Theorem 4. ■

Theorem 6 *Adaptive prioritised value iteration strongly converges to the optimal value function if the estimate of the model strongly converges to the true model.*

Proof The arguments in this proof follow those of Gullapalli and Barto (1994) for MDPs. Let V_k^* be the optimal value function at update k assuming that the estimates $\hat{P}_k$, $\hat{R}_k$, and $\hat{\gamma}_k$ are correct. Let V^* be the actual optimal value

function. Since V_k^* is a continuous function of the estimates and the estimates strongly converge to their true values, then strong convergence of V_k^* to V^* follows from the continuous mapping theorem. Consequently, for all $\epsilon > 0$ there exists a k' such that for all $k > k'$ and with probability one

$$\|V_t^* - V^*\| < (1-\alpha)\epsilon/(2\alpha). \tag{A.12}$$

Let $\{s_k\}_{k=1}^{\infty}$ be an infinite sequence of states in the order in which their value functions are updated. Hence,

$$V_{k+1}(s) = \begin{cases} BV_k(s) & \text{if } s = s_k \\ V_k(s) & \text{otherwise} \end{cases}.$$

By combining the equation above with the fact that B is a contraction (Theorem 1) and $BV_t^* = V^*$ (Theorem 2), it follows that

$$|V_{k+1}(s_k) - V^*(s_k)| \leq \alpha \|V_k - V^*\|. \tag{A.13}$$

Construct another infinite sequence, $\{k_i^s\}_{i=1}^{\infty}$, consisting of the indices $k > k'$ where $s_k = s$. By induction over i, it is possible to show that with probability one,

$$\left|V_{k_i^s+1}(s) - V_{k_i^s}^*(s)\right| \leq \alpha^{i+1} \left\|V_{k_1^s} - V_{k_1^s}^*\right\| + (1-\alpha^i)\epsilon. \tag{A.14}$$

The equation above clearly holds for $i = 1$. Assuming the equation holds for i, it is possible to show that it holds for $i+1$ using Equation A.13, the triangle inequality, Equation A.12, and the fact that $V_{k_i^s+1}(s) = V_{k_{i+1}^s}(s)$:

$$\begin{aligned}
\left|V_{k_{i+1}^s+1}(s) - V_{k_{i+1}^s}^*(s)\right| &\leq \alpha \left\|V_{k_{i+1}^s} - V_{k_{i+1}^s}^*\right\| \\
&\leq \alpha \left(\left\|V_{k_{i+1}^s} - V_{k_i^s}^*\right\| + \left\|V_{k_i^s}^* - V_{k_{i+1}^s}^*\right\|\right) \\
&< \alpha \left|V_{k_i^s+1}(s) - V_{k_{i+1}^s}(s)\right| + (1+\alpha)\epsilon \\
&< \alpha \left(\alpha^{i+1} \left\|V_{k_1^s} - V_{k_1^s}^*\right\| + (1-\alpha^i)\epsilon\right) + (1+\alpha)\epsilon \\
&= \alpha^{i+2} \left\|V_{k_1^s} - V_{k_1^s}^*\right\| + (1-\alpha^{i+1})\epsilon.
\end{aligned}$$

By taking the limit of Equation A.14 and using the fact that $\lim_{i\to\infty} V_{t_i^s}^*(s) = V^*(s)$ with probability one, one may show that with probability one,

$$\lim_{i\to\infty} \left|V_{k_i^s+1}(s) - V^*(s)\right| < \epsilon.$$

Hence, V_t strongly converges to V^*. ■

Appendix B

Random Generation of SMDPs

The experiments in Chapter 8 use a generator of random SMDPs that takes as parameters $(d, n_{\text{state}}, n_{\text{obstacle}}, r, n_{\text{reward}}, k, \epsilon)$. A randomly generated SMDP involves states that are situated in a d-dimensional unit hypercube, $[0,1)^d$. To generate a random SMDP, create n_{obstacle} d-dimensional hyperspheres with radius r and centres in the hypercube and choose n_{state} points outside the boundaries of the hypersphere obstacles. These points serve as the state space, $\mathbb{S}$. Select n_{reward} states randomly from $\mathbb{S}$, and denote this set by S_{reward}. These states are terminal, and the agent receives a lump sum of 1 when transitioning to these states.

Let the set of all possible actions, $\mathbb{A}$, consist of the standard basis vectors of the hypercube and their negation, i.e.

$$(1,0,\ldots,0), (-1,0,\ldots,0), (0,1,\ldots,0), \ldots, (0,0,\ldots,1), (0,0,\ldots,-1).$$

These actions can be thought of as cardinal directions in the hypercube. To determine $\text{succ}(s,a)$ for $s \notin S_{\text{reward}}$, find the closest k states to s[1] and of these choose the ones whose dot product with a is greater than $a \cdot s$ and the line segment to s does not intersect with one of the hyperspheres. The action set for a state s is

$$\mathbb{A}(s) = \{a \in \mathbb{A} \mid \text{succ}(s,a) \neq \emptyset\}.$$

The transition probabilities follow

$$P(s' \mid s,a) \propto \begin{cases} 0 & \text{if } s' \notin \text{succ}(s,a) \\ \frac{(s'-s)\cdot a}{\|s'-s\|} & \text{otherwise} \end{cases},$$

[1] In order to find the closest k states efficiently, it is necessary to use a data structure such as a k-d tree (Bentley, 1975; Friedman et al., 1977). Chapter 8 uses an implementation that follows the description of Moore (1990, Chapter 6).

where $\|\cdot\|$ in this context is the Euclidean norm.

The probability distribution over transition times from s to s' by action a is defined by the cumulative distribution function P_t:

$$P_t(t \mid s, a, s') \propto \begin{cases} 1 - e^{-t/\|s-s'\|} & \text{if } s' \in \text{succ}(s, a) \\ 0 & \text{otherwise} \end{cases} .$$

The probability distributions over lump-sum reward and reward rates follow the cumulative distribution functions

$$\begin{aligned} P_r(r \mid s, a, s') &= \begin{cases} \Phi\left(\frac{r-1}{\epsilon}\right) & \text{if } s' \in S_{\text{reward}} \\ 0 & \text{otherwise} \end{cases} \\ P_\rho(\rho \mid s, a, s') &= \Phi\left(\frac{\rho+1}{\epsilon}\right), \end{aligned}$$

where $\Phi(x)$ is the standard normal cumulative distribution function.

Bibliography

Aha, D. W., Kibler, D., and Albert, M. K. (1991). Instance-based learning algorithms. *Machine Learning*, 6(1):37–66. (Cited on page 90.)

Al-Ansari, M. A. and Williams, R. J. (1999). Robust, efficient, globally-optimized reinforcement learning with the Parti-game algorithm. In Kearns, M. J., Solla, S. A., and Cohn, D. A., editors, *Advances in Neural Information Processing Systems*, volume 11, pages 961–967. MIT Press. (Cited on page 45.)

Allen, J. F. (1984). Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154. (Cited on page 189.)

Alpaydın, E. (1999). Combined 5×2 cv f test for comparing supervised classification learning algorithms. *Neural Computation*, 11(8):1885–1892. (Cited on page 87.)

Alvarez, L. H. R. (2004). Stochastic forest stand value and optimal timber harvesting. *SIAM Journal on Control and Optimization*, 42(6):1972–1993. (Cited on page 39.)

Anderson, C. W. (1986). *Learning and Problem Solving with Connectionist Representations*. Doctoral thesis, Department of Computer Science, University of Massachusetts, Amherst. (Cited on page 52.)

Andre, D. and Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and the Fourteenth Conference on Innovative Applications of Artificial Intelligence*, pages 119–125. AAAI Press. (Cited on page 69.)

Anton, H. (2005). *Elementary Linear Algebra*. Wiley, 9th edition. (Cited on page 83.)

Arnold, K., Gosling, J., and Holmes, D. (2006). *The Java Programming Language*. Addison-Wesley, 4th edition. (Cited on page 138.)

Asada, M., Noda, S., Tawaratsumida, S., and Hosoda, K. (1996). Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Machine Learning*, 23(2–3):279–303. (Cited on page 21.)

Atkeson, C. G., Moore, A. W., and Schaal, S. (1997). Locally weighted learning. *Artificial Intelligence Review*, 11(1–5):11–73. (Cited on page 49.)

Atkeson, C. G. and Santamaría, J. C. (1997). A comparison of direct and model-based reinforcement learning. In *Proceedings of the International Conference on Robotics and Automation*, volume 4, pages 3557–3564. IEEE. (Cited on pages 22, 56, and 176.)

Au, M. (2005). Automatic state construction using decision trees for reinforcement learning agents. Master's thesis, Queensland University of Technology. (Cited on page 189.)

Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A., and Somenzi, F. (1997). Algebric decision diagrams and their applications. *Formal Methods in System Design*, 10(2–3):171–206. (Cited on page 36.)

Baird, L. C. (1999). *Reinforcement Learning through Gradient Descent*. Doctoral thesis, School of Computer Science, Carnegie Mellon University. (Cited on page 53.)

Baird, L. C. and Klopf, A. H. (1993). Reinforcement learning with high-dimensional, continuous actions. Technical Report WL-TR-93-1147, Wright Laboratory, Wright-Patterson Air Force Base. (Cited on page 52.)

Barto, A. G., Bradtke, S. J., and Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1–2):81–138. (Cited on pages 114, 122, and 126.)

Barto, A. G. and Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379. (Cited on pages 42 and 114.)

Baxter, J. and Bartlett, P. L. (2001). Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350. (Cited on page 17.)

Baxter, J., Tridgell, A., and Weaver, L. (2000). Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263. (Cited on page 21.)

Bell, E. T. (1934). Exponential numbers. *American Mathematical Monthly*, 41(7):411–419. (Cited on page 70.)

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press. (Cited on pages 19, 32, and 115.)

Bellman, R. (1958). On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90. (Cited on page 34.)

Ben-David, S., Eiron, N., and Long, P. M. (2003). On the difficulty of approximately maximizing agreements. *Journal of Computer and System Sciences*, 66(3):496–514. (Cited on page 86.)

Bennett, K. P., Cristianini, N., Shawe-Taylor, J., and Wu, D. (2000). Enlarging the margins in perceptron decision trees. *Machine Learning*, 41(3):295–313. (Cited on page 86.)

Benson, S. and Nilsson, N. J. (1995). Reacting, planning and learning in an autonomous agent. In Furukawa, K., Michie, D., and Muggleton, S., editors, *Machine Intelligence*, volume 14, pages 29–64. Oxford University Press. (Cited on page 56.)

Benson, S. S. (1996). *Learning Action Models for Reactive Autonomous Agents*. Doctoral thesis, Department of Computer Science, Stanford University. (Cited on page 56.)

Bensoussan, A., Sethi, S. P., Vickson, R., and Dersko, N. (1984). Stochastic production planning with production constraints. *SIAM Journal on Control and Optimization*, 22(6):920–935. (Cited on page 39.)

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517. (Cited on pages 92 and 199.)

Bertsekas, D. P. (1982). Distributed dynamic programming. *IEEE Transactions on Automatic Control*, 27(3):610–616. (Cited on page 117.)

Bertsekas, D. P. and Castañon, D. A. (1989). Adaptive aggregation methods for infinite horizon dynamic programming. *IEEE Transactions on Automatic Control*, 34(6):589–598. (Cited on page 111.)

Bertsekas, D. P. and Shreve, S. E. (1996). *Stochastic Optimal Control: The Discrete-Time Case*. Athena Scientific. (Cited on page 32.)

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific. (Cited on page 52.)

Bertsekas, D. P. and Tsitsiklis, J. N. (1997). *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific. (Cited on pages 116, 191, and 194.)

Blockeel, H. and De Raedt, L. (1998). Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1–2):285–297. (Cited on page 48.)

Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):281–300. (Cited on page 114.)

Bollig, B. and Wegener, I. (1996). Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002. (Cited on page 111.)

Bolstad, W. M. (2004). *Introduction to Bayesian Statistics*. Wiley-Interscience. (Cited on page 61.)

Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33. (Cited on page 114.)

Borkar, V. S. (1989). *Optimal Control of Diffusion Processes*. Wiley. (Cited on page 30.)

Borkar, V. S. (2005). Controlled diffusion processes. *Probability Surveys*, 2:213–244. (Cited on page 40.)

Boutilier, C., Dean, T., and Hanks, S. (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94. (Cited on page 38.)

Boutilier, C., Dearden, R., and Goldszmidt, M. (2000). Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1–2):49–107. (Cited on page 36.)

Boyan, J. and Moore, A. (1995). Generalization in reinforcement learning: Safely approximating the value function. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems*, volume 7, pages 369–376. MIT Press. (Cited on page 53.)

Bradtke, S. J. and Duff, M. O. (1995). Reinforcement learning methods for continuous-time Markov decision problems. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems*, volume 7, pages 393–400. MIT Press. (Cited on pages 27, 114, and 133.)

Brafman, R. I. and Tennenholtz, M. (2002). R-MAX—A general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231. (Cited on page 67.)

Bråten, L. E. and Tjelta, T. (2002). Semi-Markov multistate modeling of the land mobile propagation channel for geostationary satellites. *IEEE Transactions on Antennas and Propagation*, 50(12):1795–1802. (Cited on page 114.)

Bratko, I. and Urbančič, T. (1997). Transfer of control skill by machine learning. *Engineering Applications of Artificial Intelligence*, 10(1):63–71. (Cited on page 15.)

Breiman, L., Friedman, J. H., Olsen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth International Group. (Cited on pages 81, 83, 86, and 98.)

Brodley, C. E. and Utgoff, P. E. (1995). Multivariate decision trees. *Machine Learning*, 19(1):45–77. (Cited on page 86.)

Bryant, R. E. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691. (Cited on pages 106 and 111.)

Cao, X.-R. (2005). A basic formula for online policy gradient algorithms. *IEEE Transactions on Automatic Control*, 50(5):696–699. (Cited on page 17.)

Carroll, J. L., Peterson, T. S., and Owens, N. E. (2001). Memory-guided exploration in reinforcement learning. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pages 1002–1007. IEEE. (Cited on page 69.)

Chambers, R. A. and Michie, D. (1969). Man-machine co-operation on a learning task. In Parslow, R. D., Prowse, R. W., and Green, R. E., editors, *Computer Graphics: Techniques and Applications*, pages 179–186. Plenum. (Cited on page 68.)

Chapman, D. and Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In Mylopoulos, J. and Reiter, R., editors, *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 726–731. Morgan Kaufmann. (Cited on pages 42, 44, 45, 54, and 102.)

Chávez, E., Navarro, G., Baeza-Yates, R., and Marroquín, J. L. (2001). Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321. (Cited on page 92.)

Ciaccia, P. and Patella, M. (2002). Searching in metric spaces with user-defined and approximate distances. *ACM Transactions on Database Systems*, 27(4):398–437. (Cited on page 92.)

Ciardo, G., Marie, R. A., Sericola, B., and Trivedi (1990). Performability analysis using semi-Markov reward processes. *IEEE Transactions on Computers*, 39(10):1251–1264. (Cited on page 114.)

Clouse, J. A. (1996). *On Integrating Apprentice Learning and Reinforcement Learning*. Doctoral thesis, Department of Computer Science, University of Massachusetts, Amherst. (Cited on page 68.)

Cook, R. D. (1979). Influential observations in linear regression. *Journal of the American Statistical Association*, 74(365):169–174. (Cited on page 50.)

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press, 2nd edition. (Cited on pages 84 and 119.)

Cover, T. M. and Hart, P. E. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27. (Cited on pages 16 and 90.)

Cristianini, N. and Shawe-Taylor, J. (2000). *An Introduction to Support Vector Machines: And Other Kernel-Based Learning Methods*. Cambridge University Press. (Cited on page 86.)

Crook, P. A. (2006). *Learning in a State of Confusion: Employing Active Perception and Reinforcement Learning in Partially Observable Worlds*. Doctoral thesis, School of Informatics, University of Edinburgh. (Cited on page 150.)

Davidson, J. (1994). *Stochastic Limit Theory: An Introduction for Econometricians*. Oxford University Press. (Cited on page 22.)

Dean, T. and Kanazawa, K. (1989). A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150. (Cited on page 36.)

Dearden, R. and Boutilier, C. (1997). Abstraction and approximate decision-theoretic planning. *Artificial Intelligence*, 89(1–2):219–283. (Cited on page 38.)

Dearden, R., Friedman, N., and Andre, D. (1999). Model-based Bayesian exploration. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 150–159. Morgan Kaufmann. (Cited on pages 22 and 66.)

Dietterich, T. G. (2000). State abstraction in MAXQ hierarchical reinforcement learning. In Solla, S. A., Leen, T. K., and Müller, K.-R., editors, *Advances in Neural Information Processing Systems*, volume 12, pages 994–1000. MIT Press. (Cited on pages 69 and 114.)

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271. (Cited on pages 33 and 115.)

Do, M. B. and Kambhampati, S. (2001). Planning as constraint satisfaction: solving the planning graph by compiling it into CSP. *Artificial Intelligence*, 132(2):151–182. (Cited on page 114.)

Donaldson, P. E. K. (1960). Error decorrelation: A technique for matching a class of functions. In *Proceedings of the Third International Conference on Medical Electronics*, pages 173–178. Charles C. Thomas. (Cited on pages 14 and 52.)

Doya, K. (2000). Reinforcement learning in continuous time and space. *Neural Computation*, 12(1):219–245. (Cited on pages 39 and 53.)

Driessens, K. (2004). *Relational Reinforcement Learning*. Doctoral thesis, Departement Computerwetenschappen, Katholieke Universiteit, Leuven. (Cited on pages 47 and 54.)

Driessens, K. and Džeroski, S. (2004). Integrating guidance into relational reinforcement learning. *Machine Learning*, 57(3):271–304. (Cited on page 68.)

Driscoll, J. R., Gabow, H. N., Shrairman, R., and Tarjan, R. E. (1988). Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354. (Cited on pages 118 and 143.)

Duda, R. O., Hart, P. E., and Stork, D. G. (2000). *Pattern Classification*. Wiley, 2nd edition. (Cited on pages 14, 59, and 86.)

Duff, M. O. (2002). *Optimal Learning: Computational Procedures for Bayes-Adaptive Markov Decision Processes*. Doctoral thesis, Department of Computer Science, University of Massachusetts, Amherst. (Cited on page 22.)

Džeroski, S., de Raedt, L., and Driessens, K. (2001). Relational reinforcement learning. *Machine Learning*, 43(1–2):7–52. (Cited on page 47.)

Eades, P. (1984). A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160. (Cited on page 139.)

Efron, B. (1979). Bootstrap methods: Another look at the jackknife. *Annals of Statistics*, 7(1):1–26. (Cited on page 78.)

Efron, B. (1983). Estimating the error rate of a prediction rule: Improvement on cross-validation. *Journal of the American Statistical Association*, 78:316–331. (Cited on page 78.)

Efron, B. and Tibshirani, R. J. (1993). *An Introduction to the Bootstrap*. Chapman and Hall. (Cited on page 78.)

Einstein, A. (1934). On the method of theoretical physics. *Philosophy of Science*, 1(2):163–169. (Cited on page 101.)

Engel, Y. (2005). *Algorithms and Representations for Reinforcement Learning*. Doctoral thesis, Hebrew University. (Cited on page 40.)

Everitt, B. S., Landau, S., and Leese, M. (2001). *Cluster Analysis*. Oxford University Press, 4th edition. (Cited on page 76.)

Fayyad, U. M. and Irani, K. B. (1992). On the handling of continuous-valued attributes in decision tree generation. *Machine Learning*, 8(1):87–102. (Cited on page 83.)

Fikes, R., Jenkins, J., and Frank, G. (2003). JTP: A system architecture and component library for hybrid reasoning. Technical Report KSL-03-01, Knowledge Systems Laboratory, Department of Computer Science, Stanford University. (Cited on page 89.)

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208. (Cited on pages 19 and 37.)

Fisher, R. A. (1922). On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 222:309–368. (Cited on pages 22 and 59.)

Fleming, W. and Pardoux, E. (1982). Optimal control for partially observed diffusions. *SIAM Journal on Control and Optimization*, 20(2):261–285. (Cited on page 35.)

Ford, L. R. and Fulkerson, D. R. (1962). *Flows in Networks*. Princeton University Press. (Cited on page 34.)

Fox, M. and Long, D. (2003). PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124. (Cited on page 114.)

Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615. (Cited on pages 33, 119, and 143.)

Friedman, J. H., Bentley, J. L., and Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226. (Cited on pages 92 and 199.)

Friedman, N. and Singer, Y. (1999). Efficient Bayesian parameter estimation in large discrete domains. In Kearns, M. J., Solla, S. A., and Cohn, D. A., editors, *Advances in Neural Information Processing Systems*, volume 11, pages 417–423. MIT Press. (Cited on page 22.)

Fritzke, B. (1995). A growing neural gas network learns topologies. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems*, volume 7, pages 625–632. MIT Press. (Cited on page 51.)

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. (Cited on page 138.)

Gansner, E. R., Koutsofios, E., North, S. C., and Vo, K.-P. (1993). A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230. (Cited on page 139.)

Gansner, E. R. and North, S. C. (2000). An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233. (Cited on page 139.)

Gates, G. W. (1972). The reduced nearest neighbor rule. *IEEE Transactions on Information Theory*, 18(3):431–433. (Cited on page 93.)

Genesereth, M. R. and Fikes, R. E. (1992). Knowledge interchange format, version 3.0 reference manual. Technical Report KSL-92-86, Knowledge Systems Laboratory, Department of Computer Science, Stanford University. (Cited on page 89.)

Gentle, J. E. (2002). *Elements of Computational Statistics*. Springer. (Cited on page 62.)

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann. (Cited on pages 19 and 114.)

Ghavamzadeh, M. and Mahadevan, S. (2001). Continuous-time hierarchical reinforcement learning. In Brodley, C. E. and Danyluk, A. P., editors, *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 186–193. Morgan Kaufmann. (Cited on page 114.)

Giunchiglia, F. and Walsh, T. (1992). A theory of abstraction. *Artificial Intelligence*, 57(2–3):323–389. (Cited on page 42.)

Givan, R., Dean, T., and Greig, M. (2003). Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence*, 147(1–2):163–223. (Cited on page 111.)

Glover, F. and Laguna, M. (1997). *Tabu Search*. Kluwer Academic Publishers. (Cited on page 18.)

Goebel, M. C. (2005). An empirical investigation into function approximation with reinforcement learning. Master's thesis, School of Informatics, University of Edinburgh. (Cited on pages 175 and 183.)

Golub, G. H. and Van Loan, C. F. (1996). *Matrix Computations*. Johns Hopkins University Press, 3rd edition. (Cited on page 32.)

Goodman, L. A. (1960). On the exact variance of products. *Journal of the American Statistical Association*, 55(292):708–713. (Cited on page 63.)

Goodrich, M. A., Stirling, W. C., and Boer, E. R. (2000). Satisficing revisited. *Minds and Machines*, 10(1):79–109. (Cited on page 150.)

Gosavi, A., Bandla, N., and Das, T. K. (2002). A reinforcement learning approach to a single leg airline revenue management problem with multiple fare classes and overbooking. *IIE Transactions*, 34(9):729–742. (Cited on page 114.)

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java Language Specification*. Addison-Wesley, 3rd edition. (Cited on page 138.)

Gray, R. M. and Neuhoff, D. L. (1998). Quantization. *IEEE Transactions on Information Theory*, 44(6):2325–2383. (Cited on page 92.)

Guestrin, C., Koller, D., Parr, R., and Venkataraman, S. (2003). Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research*, 19:399–468. (Cited on pages 113 and 123.)

Gullapalli, V. (1990). A stochastic reinforcement learning algorithm for learning real-valued functions. *Neural Networks*, 3(6):671–692. (Cited on page 52.)

Gullapalli, V. and Barto, A. G. (1994). Convergence of indirect adaptive asynchronous value iteration algorithms. In Cowan, J. D., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems*, volume 6, pages 695–702. Morgan Kaufmann. (Cited on page 196.)

Guo, H. and Gelfand, S. B. (1992). Classification trees with neural network feature extraction. *IEEE Transactions on Neural Networks*, 3(6):923–933. (Cited on page 87.)

Hanks, S. and McDermott, D. (1994). Modeling a dynamic and uncertain world I: Symbolic and probabilistic reasoning about change. *Artificial Intelligence*, 66(1):1–55. (Cited on page 37.)

Hart, P. E. (1968). The condensed nearest neighbor rule. *IEEE Transactions on Information Theory*, 14(3):515–516. (Cited on page 93.)

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths in graphs. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107. (Cited on page 19.)

Hart, P. E., Nilsson, N. J., and Raphael, B. (1972). Correction to "A formal basis for the heuristic determination of minimum cost paths". *ACM SIGART Bulletin*, 37:28–29. (Cited on page 19.)

Hart, W. E., Krasnogor, N., and Smith, J. E., editors (2005). *Recent Advances in Memetic Algorithms*. Springer. (Cited on page 18.)

Hauskrecht, M. (2000). Value-function approximations for partially observable Markov decision processes. *Journal of Artificial Intelligence Research*, 13:33–94. (Cited on page 35.)

Hawking, S. W. (2001). *The Universe in a Nutshell*. Bantam Books. (Cited on page 185.)

Haykin, S. S. (1999). *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 2nd edition. (Cited on page 15.)

Heath, D. (1992). *A Geometric Framework for Machine Learning*. Doctoral thesis, Department of Computer Science, Johns Hopkins University. (Cited on page 86.)

Heath, D., Kasif, S., and Salzberg, S. (1993). Induction of oblique decision trees. In Bajcsy, R., editor, *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, volume 2, pages 1002–1007. Morgan Kaufmann. (Cited on page 86.)

Herbrich, R. (2002). *Learning Kernel Classifiers: Theory and Algorithms*. MIT Press. (Cited on page 86.)

Hjaltason, G. R. and Samet, H. (2003). Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580. (Cited on page 92.)

Hoey, J., St-Aubin, R., Hu, A., and Boutilier, C. (2000). Optimal and approximate stochastic planning using decision diagrams. Technical Report TR-00-05, Department of Computer Science, University of British Columbia. (Cited on page 36.)

Höffgen, K. U., Simon, H. U., and Van Horn, K. S. (1995). Robust trainability of single neurons. *Journal of Computer and System Sciences*, 50(1):114–125. (Cited on page 86.)

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press. (Cited on page 18.)

Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. MIT Press. (Cited on page 32.)

Howard, R. A. (1971). *Dynamic Probabilistic Systems*, volume 2 of *Series in decision and control*. Wiley. (Cited on pages 27 and 60.)

Hyafil, L. and Rivest, R. L. (1976). Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17. (Cited on page 112.)

Jain, A. K., Dubes, R. C., and Chen, C.-C. (1987). Bootstrap techniques for error estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 9(5):628–633. (Cited on page 78.)

James, M. R. (2005). *Using Predictions for Planning and Modeling in Stochastic Environments*. Doctoral thesis, Department of Electrical Engineering and Computer Science, University of Michigan. (Cited on page 34.)

Jaynes, E. T. (2003). *Probability Theory: The Logic of Science*. Cambridge University Press. (Cited on page 61.)

Jong, N. K. and Stone, P. (2005). State abstraction discovery from irrelevant state variables. In Kaelbling, L. P. and Saffiotti, A., editors, *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 752–757. Professional Book Center. (Cited on page 42.)

Kaelbling, L. P. (1993). *Learning in Embedded Systems*. MIT Press. (Cited on page 66.)

Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134. (Cited on page 35.)

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285. (Cited on pages 20 and 27.)

Kalantari, I. and McDonald, G. (1983). A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, 9(5):631–634. (Cited on page 92.)

Kamada, T. and Kawai, S. (1989). An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15. (Cited on page 139.)

Kandel, E. R. (2001). The molecular biology of memory storage: A dialogue between genes and synapses. *Science*, 294(5544):1030–1038. (Cited on page 190.)

Kautz, H. and Selman, B. (1992). Planning as satisfiability. In Neumann, B., editor, *Proceedings of the Tenth European Conference on Artificial Intelligence*, pages 359–363. Wiley. (Cited on page 114.)

Kearns, M. and Singh, S. (2002). Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2–3):209–232. (Cited on page 67.)

Kearns, M. J. and Singh, S. P. (1999). Finite-sample convergence rates for Q-learning and indirect algorithms. In Kearns, M. J., Solla, S. A., and Cohn, D. A., editors, *Advances in Neural Information Processing Systems*, volume 11, pages 996–1002. MIT Press. (Cited on page 22.)

Khardon, R. (1999). Learning action strategies for planning domains. *Artificial Intelligence*, 113(1–2):125–148. (Cited on page 16.)

Khoo, A. and Zubek, R. (2002). Applying inexpensive AI techniques to computer games. *IEEE Intelligent Systems and Their Applications*, 17(4):48–53. (Cited on page 147.)

Kim, K.-E. and Dean, T. (2002). Solving factored MDPs with large action space using algebraic decision diagrams. In Ishizuka, M. and Sattar, A., editors, *Proceedings of the Seventh Pacific Rim International Conference on Artificial Intelligence*, volume 2417 of *Lecture Notes in Computer Science*, pages 80–89. Springer-Verlag. (Cited on page 36.)

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680. (Cited on page 18.)

Kloeden, P. E. and Platen, E. (1999). *Numerical Solution of Stochastic Differential Equations*. Springer. (Cited on pages 30 and 31.)

Kochenderfer, M. J. (2003). Evolving hierarchical and recursive teleo-reactive programs through genetic programming. In Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., and Costa, E., editors, *Proceedings of the Sixth European Conference on Genetic Programming*, volume 2610 of *Lecture Notes in Computer Science*, pages 83–92. Springer Verlag. (Cited on page 18.)

Kochenderfer, M. J. (2005). Adaptive modeling and planning for reactive agents. In Veloso, M. M. and Kambhampati, S., editors, *Proceedings of the Twentieth National Conference on Artificial Intelligence and the Seventeenth Annual Conference on Innovative Applications of Artificial Intelligence*, pages 1648–1649. AAAI Press. (Cited on page iv.)

Kochenderfer, M. J. and Hayes, G. (2005a). Adaptive partitioning of state spaces using decision graphs for real-time modeling and planning. In Bulitko, V. and Koenig, S., editors, *Proceedings of the Workshop on Planning and Learning in A Priori Unknown and Dynamic Domains*, pages 9–15. International Joint Conferences on Artificial Intelligence. (Cited on page iv.)

Kochenderfer, M. J. and Hayes, G. (2005b). Modeling and planning in large state and action spaces. In Bulitko, V. and Koenig, S., editors, *Proceedings of the Workshop on Planning and Learning in A Priori Unknown and Dynamic Domains*, pages 16–22. International Joint Conferences on Artificial Intelligence. (Cited on page iv.)

Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1137–1145. Morgan Kaufmann. (Cited on page 78.)

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69. (Cited on pages 50 and 51.)

Kohonen, T. (2001). *Self-Organizing Maps*. Springer. (Cited on page 50.)

Korn, R. and Kraft, H. (2001). A stochastic control approach to portfolio problems with stochastic interest rates. *SIAM Journal on Control and Optimization*, 40(4):1250–1269. (Cited on page 39.)

Koza, J. R. (1992). *Genetic Programming: On The Programming of Computers by Means of Natural Selection*. MIT Press. (Cited on page 18.)

Krętowski, M. (2004). An evolutionary algorithm for oblique decision tree induction. In Rutkowski, L., Siekmann, J., Tadeusiewicz, R., and Zadeh, L. A., editors, *Proceedings of the Seventh International Conference on Artificial Intelligence and Soft Computing*, volume 3070 of *Lecture Notes in Computer Science*, pages 432–437. Springer. (Cited on page 87.)

Kumar, P. and Becker, A. (1982). A new family of optimal adaptive controllers for Markov chains. *IEEE Transactions on Automatic Control*, 27(1):137–146. (Cited on page 122.)

Kumar, P. R. (1985). A survey of some results in stochastic adaptive control. *SIAM Journal on Control and Optimization*, 23(3):329–380. (Cited on pages 21 and 122.)

Kushmerick, N., Hanks, S., and Weld, D. S. (1995). An algorithm for probabilistic planning. *Artificial Intelligence*, 76(76):239–286. (Cited on pages 38 and 113.)

Kushner, H. (1990). Numerical methods for stochastic control problems in continuous time. *SIAM Journal on Control and Optimization*, 28(5):999–1048. (Cited on page 31.)

Kushner, H. J. and Dupuis, P. (2001). *Numerical methods for stochastic control problems in continuous time*. Springer, 2nd edition. (Cited on page 30.)

Lachenbruch, P. A. and Mickey, M. R. (1968). Estimation of error rates in discriminant analysis. *Technometrics*, 10(1):1–11. (Cited on page 78.)

Laud, A. and DeJong, G. (2003). The influence of reward on the speed of reinforcement learning: An analysis of shaping. In Fawcett, T. and Mishra, N., editors, *Proceedings of the Twentieth International Conference on Machine Learning*, volume 1, pages 440–447. AAAI Press. (Cited on page 69.)

Lee, J. L. C., Everitt, B. J., and Thomas, K. L. (2004). Independent cellular processes for hippocampal memory consolidation and reconsolidation. *Science*, 304(5672):839–843. (Cited on page 190.)

Lehmann, E. L. and Romano, J. P. (2005). *Testing Statistical Hypotheses*. Springer, 3rd edition. (Cited on page 44.)

Likhachev, M. and Koenig, S. (2003). Speeding up the Parti-game algorithm. In Becker, S., Thrun, S., and Obermayer, K., editors, *Advances in Neural Information Processing Systems*, volume 15, pages 1563–1570. MIT Press. (Cited on page 45.)

Lin, L.-J. (1991). Programming robots using reinforcement learning and teaching. In Dean, T. L. and McKeown, K., editors, *Proceedings of the Ninth National Conference on Artificial Intelligence*, volume 2, pages 781–786. AAAI Press. (Cited on page 68.)

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3–4):293–321. (Cited on page 52.)

Lin, L.-J. and Mitchell, T. M. (1992). Memory approaches to reinforcement learning in non-Markovian domains. Technical Report CMU-CS-92-138, School of Computer Science, Carnegie Mellon University. (Cited on page 189.)

Littman, M. L. (1994). Memoryless policies: Theoretical limitations and practical results. In Cliff, D., Husbands, P., Meyer, J.-A., and Wilson, S. W., editors, *Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pages 238–245. MIT Press. (Cited on page 189.)

Littman, M. L. (1996). *Algorithms for Sequential Decision Making*. Doctoral thesis, Department of Computer Science, Brown University. (Cited on page 27.)

Littman, M. L. (1997). Probabilistic propositional planning: Representations and complexity. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and the Ninth Innovative Applications of Artificial Intelligence Conference*, pages 748–754. AAAI Press. (Cited on page 38.)

Littman, M. L., Sutton, R. S., and Singh, S. P. (2002). Predictive representations of state. In Dietterich, T. G., Becker, S., and Ghahramani, Z., editors, *Advances in Neural Information Processing Systems*, volume 14, pages 1555–1561. MIT Press. (Cited on page 34.)

Loch, J. and Singh, S. (1998). Using eligibility traces to find the best memoryless policy in partially observable markov decision processes. In Shavlik, J. W., editor, *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 323–331. Morgan Kaufmann. (Cited on page 189.)

López de Mántaras, R. (1991). A distance-based attribute selection measure for decision tree induction. *Machine Learning*, 6(1):81–92. (Cited on page 81.)

Luenberger, D. G. (1969). *Optimization by Vector Space Methods*. Wiley. (Cited on page 194.)

Madani, O., Hanks, S., and Condon, A. (2003). On the undecidability of probabilistic planning and related stochastic optimization problems. *Artificial Intelligence*, 147(1–2):5–34. (Cited on page 35.)

Mahadevan, S. (1992). Enhancing transfer in reinforcement learning by building stochastic models of robot actions. In Sleeman, D. H. and Edwards, P., editors, *Proceedings of the Ninth International Workshop on Machine Learning*, pages 290–299. Morgan Kaufmann. (Cited on page 22.)

Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22(1–3):159–195. (Cited on page 27.)

Mahadevan, S., Marchalleck, N., Das, T. K., and Gosavi, A. (1997). Self-improving factory simulation using continuous-time average-reward reinforcement learning. In Fisher, D. H., editor, *Proceedings of the Fourteenth International Confrence on Machine Learning*, pages 202–210. Morgan Kaufmann. (Cited on page 114.)

Marsland, S., Shapiro, J., and Nehmzow, U. (2002). A self-organising network that grows when required. *Neural Networks*, 15(8–9):1041–1058. (Cited on page 51.)

Mataric, M. J. (1994). Reward functions for accelerated learning. In Cohen, W. W. and Hirsh, H., editors, *Proceedings of the Eleventh International Conference on Machine Learning*, pages 181–189. Morgan Kaufmann. (Cited on page 69.)

McCallum, A. K. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. Doctoral thesis, Department of Computer Science, University of Rochester. (Cited on pages 35, 42, 45, 46, 82, 100, 102, 177, and 189.)

McCallum, A. K. (1996a). Learning to use selective attention and short-term memory in sequential tasks. In Maes, P., Mataric, M. J., Meyer, J.-A., Pollack, J., and Wilson, S. W., editors, *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pages 315–324. MIT Press. (Cited on pages 20, 100, 102, 177, and 189.)

McCallum, R. A. (1996b). Hidden state and reinforcement learning with instance-based state identification. *IEEE Transactions on Systems, Man and Cybernetics, Part B*, 26(3):464–473. (Cited on pages 189 and 190.)

McDermott, D. (2000). The 1998 AI Planning Systems competition. *AI Magazine*, 21(2):35–55. (Cited on page 114.)

McMahan, H. B. and Gordon, G. J. (2005). Generalizing Dijkstra's algorithm and Gaussian elimination for solving MDPs. Technical Report CMU-CS-05-127, Computer Science Department, Carnegie Mellon University. (Cited on page 115.)

Meeker, W. Q. and Escobar, L. A. (1998). *Statistical Methods for Reliability Data*. Wiley. (Cited on page 63.)

Metoyer, R. A. and Hodgins, J. K. (2000). Animating athletic motion planning by example. In Fels, S. and Poulin, P., editors, *Proceedings of the Graphics Interface Conference*, pages 61–68. Canadian Human-Computer Communications Society. (Cited on page 16.)

Meuleau, N. and Bourgine, P. (1999). Exploration of multi-state environments: local measures and back-propagation of uncertainty. *Machine Learning*, 35(2):117–154. (Cited on page 66.)

Michie, D. and Camacho, R. (1994). Building symbolic representations of intuitive real-time skills from performance data. In Furukawa, K., Michie, D., and Muggleton, S., editors, *Machine Intelligence*, volume 13, pages 385–418. Clarendon. (Cited on page 15.)

Michie, D. and Chambers, R. A. (1968a). BOXES: An experiment in adaptive control. In Dale, E. and Michie, D., editors, *Machine Intelligence*, volume 2, pages 137–152. Oliver and Boyd. (Cited on page 43.)

Michie, D. and Chambers, R. A. (1968b). 'Boxes' as a model of pattern-formation. In Waddington, C. H., editor, *Towards a Theoretical Biology*, volume 1, pages 206–215. Edinburgh University Press. (Cited on page 43.)

Millán, J., Posenato, D., and Dedieu, E. (2002). Continuous-action Q-learning. *Machine Learning*, 49(2-3):247–265. (Cited on page 51.)

Milligan, G. W. and Cooper, M. C. (1985). An examination of procedures for detecting the number of clusters in a data set. *Psychometrika*, 50:159–179. (Cited on page 76.)

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press. (Cited on page 18.)

Moore, A. W. (1990). *Efficient Memory-Based Learning for Robot Control*. Doctoral thesis, University of Cambridge. (Cited on pages 93 and 199.)

Moore, A. W. and Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13(1):103–130. (Cited on pages 22, 56, 67, 71, 115, 117, 118, 119, 120, 122, 123, 126, 127, 133, 146, 175, and 176.)

Moore, A. W. and Atkeson, C. G. (1995). The Parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21(3):199–233. (Cited on pages 44, 45, 53, and 77.)

Moore, D. W. (1992). *Simplical Mesh Generation with Applications*. Doctoral thesis, Cornell University. (Cited on page 47.)

Muggleton, S. and De Raedt, L. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19–20:629–679. (Cited on page 56.)

Munos, R. (1997). *L'Apprentissage par Renforcement: Etude du Cas Continu*. Doctoral thesis, Ecole des Hautes Etudes en Sciences Sociales. (Cited on page 39.)

Munos, R. (2000). A study of reinforcement learning in the continuous case by the means of viscosity solutions. *Machine Learning*, 40(3):265–299. (Cited on pages 71 and 99.)

Munos, R. and Bourgine, P. (1998). Reinforcement learning for continuous stochastic control problems. In Jordan, M. I., Kearns, M. J., and Solla, S. A., editors, *Advances in Neural Information Processing Systems*, volume 10, pages 1029–1035. MIT Press. (Cited on pages 39 and 99.)

Munos, R. and Moore, A. (2002). Variable resolution discretization in optimal control. *Machine Learning*, 49(2–3):291–323. (Cited on pages 39 and 47.)

Murthy, S. K., Kasif, S., and Salzberg, S. (1994). A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research*, 2:1–32. (Cited on pages 81 and 86.)

Neapolitan, R. E. (2004). *Learning Bayesian Networks*. Pearson Prentice Hall. (Cited on page 36.)

Newton, I. (1687). *Philosophiæ Naturalis Principia Mathematica*. Jussu Societatis Regiæ ac typis Josephi Streater. (Cited on page 73.)

Ng, A. Y., Harada, D., and Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In Bratko, I. and Džeroski, S., editors, *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287. Morgan Kaufmann. (Cited on page 69.)

Ng, A. Y. and Jordan, M. I. (2000). PEGASUS: A policy search method for large MDPs and POMDPs. In Boutilier, C. and Goldszmidt, M., editors, *Proceedings of the Sixteenth Conference in Uncertainty in Artificial Intelligence*, pages 405–415. Morgan Kaufmann. (Cited on page 183.)

Nilsson, N. (1992). Toward agent programs with circuit semantics. Technical Report STAN-CS-92-1412, Department of Computer Science, Stanford University. (Cited on page 103.)

Nilsson, N. J. (1984). Shakey the robot. Technical Report 323, AI Center, SRI International. (Cited on page 19.)

Nilsson, N. J. (1994). Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158. (Cited on page 13.)

Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann. (Cited on page 11.)

Nilsson, N. J. (2000). Learning strategies for mid-level robot control: Some preliminary considerations and results. `http://ai.stanford.edu/~nilsson`. (Cited on pages 103 and 111.)

Nilsson, N. J. (2001). Teleo-reactive programs and the triple-tower architecture. *Electronic Transactions on Artificial Intelligence*, 5:99–110. (Cited on page 13.)

Nye, M. J., editor (2002). *The Modern Physical and Mathematical Sciences*, volume 5 of *The Cambridge History of Science*. Cambridge University Press. (Cited on page 102.)

Øksendal, B. K. (2003). *Stochastic Differential Equations: An Introduction with Applications*. Springer, 6th edition. (Cited on pages 30 and 31.)

Olukotun, K. and Hammond, L. (2005). The future of microprocessors. *Queue*, 3(7):26–29. (Cited on page 190.)

Ormoneit, D. and Sen, Ś. (2002). Kernel-based reinforcement learning. *Machine Learning*, 49(2–3):161–178. (Cited on page 54.)

Papadopoulos, A. N. and Manolopoulos, Y. (2005). *Nearest Neighbor Search: A Database Perspective*. Springer. (Cited on page 92.)

Pareigis, S. (1997). Multi-grid methods for reinforcement learning in controlled diffusion processes. In *Advances in Neural Information Processing Systems*. MIT Press. (Cited on page 39.)

Parr, R. E. (1998). *Hierarchical Control and Learning for Markov Decision Processes*. Doctoral thesis, University of California, Berkeley. (Cited on pages 69 and 114.)

Patterson, D. A. and Hennessy, J. L. (2005). *Computer Organization and Design: The Hardware/Software Interface*. Elsevier, 3rd edition. (Cited on page 190.)

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann. (Cited on page 36.)

Peng, J. and Williams, R. J. (1993). Efficient learning and planning within the Dyna framework. *Adaptive Behavior*, 1(4):437–454. (Cited on pages 22, 56, 115, and 176.)

Peng, J. and Williams, R. J. (1996). Incremental multi-step Q-learning. *Machine Learning*, 22(1–3):283–290. (Cited on page 52.)

Perman, M., Senegacnik, A., and Tuma, M. (1997). Semi-Markov models with an application to power-plant reliability analysis. *IEEE Transactions on Reliability*, 46(4):526–532. (Cited on page 114.)

Pomerleau, D. A. (1993). *Neural Network Perception for Mobile Robot Guidance*. Kluwer Academic Publishers. (Cited on page 15.)

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley. (Cited on pages 27 and 60.)

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106. (Cited on pages 81, 83, and 98.)

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers. (Cited on pages 15, 83, 98, and 171.)

Ramon, J. (2002). *Clustering and Instance Based Learning in First Order Logic*. Doctoral thesis, Departement Computerwetenschappen, Katholieke Universiteit, Leuven. (Cited on page 92.)

Randløv, J. and Alstrøm, P. (1998). Learning to drive a bicycle using reinforcement learning and robot shaping. In Shavlik, J. W., editor, *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 463–471. Morgan Kaufmann. (Cited on page 69.)

Rasmussen, C. E. (2004). Gaussian processes in machine learning. In Bousquet, O., von Luxburg, U., and Rätsch, G., editors, *Advanced Lectures on Machine Learning*, volume 3176 of *Lecture Notes in Computer Science*, pages 63–71. Springer. (Cited on page 40.)

Rasmussen, C. E. and Kuss, M. (2004). Gaussian processes in reinforcement learning. In Thrun, S., Saul, L. K., and Schölkopf, B., editors, *Advances in Neural Information Processing Systems*, volume 16, pages 751–759. MIT Press. (Cited on page 40.)

Reynolds, S. I. (2000). Adaptive resolution model-free reinforcement learning: Decision boundary partitioning. In Langley, P., editor, *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 783–790. Morgan Kaufmann. (Cited on page 42.)

Rivest, R. L. (1987). Learning decision lists. *Machine Learning*, 2(3):229–246. (Cited on page 16.)

Rota, G.-C. (1964). The number of partitions of a set. *American Mathematical Monthly*, 71(5):498–504. (Cited on page 70.)

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323:533–536. (Cited on pages 15 and 52.)

Rummery, G. A. (1995). *Problem Solving with Reinforcement Learning*. Doctoral thesis, Department of Engineering, University of Cambridge. (Cited on page 173.)

Rummery, G. A. and Niranjan, M. (1994). On-line Q-learning using connectionist systems. Technical report, Department of Engineering, University of Cambridge. (Cited on page 52.)

Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition. (Cited on page 11.)

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229. (Cited on page 52.)

Samuel, A. L. (1967). Some studies in machine learning using the game of checkers. II—Recent progress. *IBM Journal of Research and Development*, 11(6):601–617. (Cited on page 52.)

Santamaría, J. C., Sutton, R. S., and Ram, A. (1997). Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 6(2):163–217. (Cited on pages 52 and 53.)

Schölkopf, B. and Smola, A. J. (2000). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press. (Cited on page 87.)

Schweitzer, P. J., Puterman, M. L., and Kindle, K. W. (1985). Iterative aggregation-disaggregation procedures for discounted semi-Markov reward processes. *Operations Research*, 33(3):589–605. (Cited on page 111.)

Scott, G. J. and Shyu, C.-R. (2003). EBS k-d tree: An entropy balanced statistical k-d tree for image databases with ground-truth labels. In Bakker, E. M., Huang, T. S., Lew, M. S., Sebe, N., and Zhou, X., editors, *Second International Conference on Image and Video Retrieval*, volume 2728 of *Lecture Notes in Computer Science*, pages 467–476. Springer. (Cited on page 93.)

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656. (Cited on pages 16, 78, and 81.)

Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press. (Cited on page 87.)

Shyu, C.-R., Chi, P.-H., Scott, G. J., and Xu, D. (2004). ProteinDBS: A real-time retrieval system for protein structure comparison. *Nucleic Acids Research*, 32(Web Server issue):W572–W575. (Cited on page 93.)

Simon, H. A. (1956). Rational choice and the structure of the environment. *Psychological Review*, 63:129–138. (Cited on page 150.)

Simons, J., Brussel, H., de Schutter, J., and Verhaert, J. (1982). A self-learning automaton with variable resolution for high precision assembly by industrial robots. *IEEE Transactions on Automatic Control*, 27(5):1109–1113. (Cited on page 44.)

Simpson, J. A. and Weiner, E. S. C., editors (1989). *The Oxford English Dictionary*. Clarendon Press, 2nd edition. (Cited on page 1.)

Smart, W. D. (2002). *Making Reinforcement Learning Work on Real Robots*. Doctoral thesis, Department of Computer Science, Brown University. (Cited on pages 49 and 99.)

Smart, W. D. and Kaelbling, L. P. (2000). Practical reinforcement learning in continuous spaces. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 903–910. Morgan Kaufmann. (Cited on pages 49, 68, and 99.)

Smith, A. J. (2002a). Applications of the self-organising map to reinforcement learning. *Neural Networks*, 15(8–9):1107–1124. (Cited on pages 50, 57, and 99.)

Smith, A. J. (2002b). *Dynamic Generalisation of Continuous Action Spaces in Reinforcement Learning: A Neurally Inspired Approach*. Doctoral thesis, Division of Informatics, University of Edinburgh. (Cited on pages 50 and 99.)

Sondik, E. J. (1978). The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research*, 26(2):282–304. (Cited on page 35.)

Strens, M. J. A. (2000). A Bayesian framework for reinforcement learning. In Langley, P., editor, *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 943–950. Morgan Kaufmann. (Cited on page 22.)

Sutton, R. S. (1984). *Temporal Credit Assignment in Reinforcement Learning*. Doctoral thesis, Department of Computer Science, University of Massachusetts, Amherst. (Cited on page 52.)

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44. (Cited on pages 33 and 114.)

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224. Morgan Kaufmann. (Cited on page 67.)

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press. (Cited on pages 20, 32, 33, 39, and 173.)

Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In Solla, S. A., Leen, T. K., and Müller, K.-R., editors, *Advances in Neural Information Processing Systems*, volume 12, pages 1057–1063. MIT Press. (Cited on page 17.)

Sutton, R. S., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1–2):181–211. (Cited on pages 42, 69, and 114.)

Tan, P. J. and Dowe, D. L. (2004). MML inference of oblique decision trees. In Webb, G. I. and Yu, X., editors, *Proceedings of the Seventeenth Australian Joint Conference on Artificial Intelligence*, volume 3339 of *Lecture Notes in Computer Science*, pages 1082–1088. Springer. (Cited on page 87.)

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8(3–4):257–277. (Cited on page 53.)

Tesauro, G. (2002). Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(134):181–199. (Cited on page 21.)

Thrun, S. (2000). Monte Carlo POMDPs. In Solla, S. A., Leen, T. K., and Müller, K.-R., editors, *Advances in Neural Information Processing Systems*, volume 12, pages 1064–1070. MIT Press. (Cited on page 35.)

Thrun, S. and Schwartz, A. (1993). Issues in using function approximation for reinforcement learning. In Mozer, M., Smolensky, P., Touretzky, D., Elman, J., and Weigend, A., editors, *Proceedings of the Fourth Connectionist Models Summer School*, pages 255–263. Lawrence Erlbaum Associates. (Cited on page 53.)

Thrun, S. B. (1992). The role of exploration in learning control. In White, D. A. and Sofge, D. A., editors, *Handbook for Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, pages 527–559. Van Nostrand Reinhold. (Cited on pages 21 and 66.)

Tollis, I. G., Di Battista, G., Eades, P., and Tamassia, R. (1999). *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall. (Cited on page 139.)

Tsitsiklis, J. N. (1994). Asynchronous stochastic approximation and Q-learning. *Machine Learning*, 16(3):185–202. (Cited on page 33.)

Utgoff, P. E. (1989). Perceptron trees: A case study in hybrid concept representations. *Connection Science*, 1(4):337–391. (Cited on page 86.)

Utgoff, P. E., Berkman, N. C., and Clouse, J. A. (1997). Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29(1):5–44. (Cited on page 16.)

Uther, W. T. B. (2002). *Tree Based Hierarchical Reinforcement Learning*. Doctoral thesis, Computer Science Department, School of Computer Science, Carnegie Mellon University. (Cited on pages 46, 71, 78, 83, 177, 178, and 180.)

Uther, W. T. B. and Veloso, M. M. (1998). Tree based discretization for continuous state space reinforcement learning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and the Tenth Conference on Innovative Applications of Artificial Intelligence*, pages 769–775. AAAI Press. (Cited on pages 177 and 178.)

Uther, W. T. B. and Veloso, M. M. (2003). TTree: Tree-based state generalization with temporally abstract actions. In Alonso, E., Kudenko, D., and Kazakov, D., editors, *Adaptive Agents and Multi-Agent Systems: Adaptation and Multi-Agent Learning*, volume 2636 of *Lecture Notes in Computer Science*, pages 266–296. Springer. (Cited on pages 71 and 180.)

Van Otterlo, M. (2005). A survey of reinforcement learning in relational domains. Technical Report TR-CTIT-05-31, Department of Computer Science, University of Twente. (Cited on page 48.)

Vollbrecht, H. (2003). *Hierarchical Reinforcement Learning in Continuous State Spaces*. Doctoral thesis, Abteilung Neuroinformatik, Universität Ulm. (Cited on page 42.)

Walker, J., Garrett, S., and Wilson, M. (2003). Evolving controllers for real robots: A survey of the literature. *Adaptive Behavior*, 11(3):179–203. (Cited on page 19.)

Watkins, C. J. and Dayan, P. (1992). Technical note: Q-learning. *Machine Learning*, 8(3–4):279–292. (Cited on page 33.)

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. Doctoral thesis, King's College, University of Cambridge. (Cited on pages 33, 52, 69, 114, and 173.)

Watkins, D. S. (2002). *Fundamentals of Matrix Computations*. Wiley-Interscience, 2nd edition. (Cited on page 32.)

Weiss, S. M. (1991). Small sample error rate estimation for k-NN classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(3):285–289. (Cited on page 78.)

Weizenbaum, J. (1966). ELIZA—A computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45. (Cited on page 13.)

White, D. J. (1993). *Markov Decision Processes*. John Wiley and Sons. (Cited on page 27.)

Whitehead, S. D. (1991). A complexity analysis of cooperative mechanisms in reinforcement learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, volume 2, pages 607–613. AAAI Press. (Cited on page 67.)

Whitehead, S. D. and Lin, L.-J. (1995). Reinforcement learning of non-Markov decision processes. *Artificial Intelligence*, 73(1–2):271–306. (Cited on page 35.)

Widrow, B. and Hoff, M. E. (1960). Adaptive switching circuits. *IRE Western Electronic Show and Convention Record*, 4(4):96–104. (Cited on page 14.)

Wiering, M. (1999). *Explorations in Efficient Reinforcement Learning*. Doctoral thesis, Faculteit der Wiskunde, Informatica, Natuurkunde en Sterrenkunde, Universiteit van Amsterdam. (Cited on pages 21, 115, 118, 119, 121, 122, 123, 126, 127, and 133.)

Wiering, M. and Schmidhuber, J. (1997). HQ-learning. *Adaptive Behavior*, 6(2):219–246. (Cited on page 189.)

Wiering, M. and Schmidhuber, J. (1998). Efficient model-based exploration. In Pfeiffer, R., Blumberg, B., Meyer, J. A., and Wilson, S. W., editors, *Proceedings of the Fifth International Conference on the Simulation of Adaptive Behavior*, pages 223–228. MIT Press. (Cited on page 66.)

Wiewiora, E. (2003). Potential-based shaping and Q-value initialization are equivalent. *Journal of Artificial Intelligence Research*, 19:205–208. (Cited on page 69.)

Wiewiora, E., Cottrell, G., and Elkan, C. (2003). Principled methods for advising reinforcement learning agents. In Fawcett, T. and Mishra, N., editors, *Proceedings of the Twentieth International Conference on Machine Learning*, volume 2, pages 792–799. AAAI Press. (Cited on page 69.)

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83. (Cited on pages 150 and 178.)

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3–4):229–256. (Cited on page 17.)

Williams, R. J. and Baird, L. C. (1993). Tight performance bounds on greedy policies based on imperfect value functions. Technical Report NU-CCS-93-14, College of Computer Science, Northeastern University. (Cited on page 194.)

Wilson, D. R. and Martinez, T. R. (1997). Improved heterogeneous distance functions. *Journal of Artificial Intelligence Research*, 6:1–34. (Cited on page 92.)

Wingate, D. and Seppi, K. D. (2004). P3VI: A partitioned, prioritized, parallel value iterator. In Brodley, C. E., editor, *Proceedings of the Twenty-first International Conference on Machine Learning*, pages 109–116. ACM Press. (Cited on page 191.)

Wingate, D. and Seppi, K. D. (2005). Prioritization methods for accelerating MDP solvers. *Journal of Machine Learning Research*, 6:851–881. (Cited on pages 115 and 134.)

Xu, R. and Wunsch, D. (2005). Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678. (Cited on page 76.)

Yianilos, P. N. (1993). Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM*

Symposium on Discrete Algorithms, pages 311–321. Society for Industrial and Applied Mathematics. (Cited on page 92.)

Yıldız, O. T. and Alpaydın, E. (2001). Omnivariate decision trees. *IEEE Transactions on Neural Networks*, 12(6):1539–1546. (Cited on page 87.)

Yoshida, W. and Ishii, S. (2005). Model-based reinforcement learning: A computational model and an fMRI study. *Neurocomputing*, 63:253–269. (Cited on page 23.)

Younes, H. L. S., Littman, M. L., Weissman, D., and Asmuth, J. (2005). The first probabilistic track of the International Planning Competition. *Journal of Artificial Intelligence Research*, 24:851–887. (Cited on page 37.)

Zettlemoyer, L. S., Pasula, H. M., and Kaelbling, L. P. (2005). Learning planning rules in noisy stochastic worlds. In Veloso, M. M. and Kambhampati, S., editors, *Proceedings of the Twentieth National Conference on Artificial Intelligence and the Seventeenth Conference on Innovative Applications of Artificial Intelligence*, pages 911–918. AAAI Press. (Cited on page 37.)

Notation

$\mathbb{A}$	set of all actions, page 25
$\mathbb{A}(s)$	set of all actions available from state s, page 25
$\mathcal{A}(S)$	partition of the action space for state region S, page 57
β	continuous compound discount rate, page 26
H	priority queue, page 118
π	stationary deterministic policy, page 26
$\pi(s)$	greedy action at state s according to policy π, page 26
$P_\rho(\rho \mid s, a, s')$	probability that the agent receives reward at a rate less than or equal to ρ while transitioning from state s to state s' by action a, page 28
$P_r(r \mid s, a, s')$	probability that the agent receives a lump-sum reward less than or equal to r while transitioning from state s to state s' by action a, page 28
$P(s' \mid s, a)$	probability that taking action a in state s will result in a transition to state s', page 27
$P_t(t \mid s, a, s')$	probability that the transition from s to s' by action a completes within time t, page 27
$r(s, a, s')$	expected lump-sum reward when transitioning from state s to state s' by action a, page 28
$\rho(s, a, s')$	expected reward rate when transitioning from state s to state s' by action a, page 28
$\mathbb{S}$	set of all states, page 25
$\mathcal{S}$	partition of the state space, page 57
s_0	absorbing state, page 65
V	value function, page 26
$V(s)$	value, or expected discounted return, of state s, page 26