Hierarchical Control and Learning for Markov Decision Processes

by

Ronald Edward Parr

A.B. (Princeton University) 1990

A dissertation submitted in partial satisfaction of the requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Stuart Russell, Chair
Professor Jitendra Malik
Professor Thomas Marschak

1998

The dissertation of Ronald Edward Parr is approved:

Chair	Date
	Date
	Date

University of California at Berkeley

1998

Abstract

Hierarchical Control and Learning
for
Markov Decision Processes

by

Ronald Edward Parr
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Stuart Russell, Chair

This dissertation investigates the use of hierarchy and problem decomposition as a means of solving large, stochastic, sequential decision problems. These problems are framed as Markov decision problems (MDPs). The new technical content of this dissertation begins with a discussion of the concept of temporal abstraction. Temporal abstraction is shown to be equivalent to the transformation of a policy defined over a region of an MDP to an action in a semi-Markov decision problem (SMDP). Several algorithms are presented for performing this transformation efficiently.

This dissertation introduces the HAM method for generating hierarchical, temporally abstract actions. This method permits the partial specification of abstract actions in a way that corresponds to an abstract plan or strategy. Abstract actions specified as HAMs can be optimally refined for new tasks by solving a *reduced* SMDP. The formal results show that traditional MDP algorithms can be used to optimally refine HAMs for new tasks. This can be achieved in much less time than it would take to learn a new policy for the task from scratch.

HAMs complement some novel decomposition algorithms that are presented in this dissertation. These algorithms work by constructing a cache of policies for different regions of the MDP and then optimally combining the cached solution to produce a global solution that is within provable bounds of the optimal solution.

Together, the methods developed in this dissertation provide important tools for producing good policies for large MDPs. Unlike some ad-hoc methods, these methods provide strong formal guarantees. They use prior knowledge in a principled way, and they reduce larger MDPs into smaller ones while maintaining a well-defined relationship between the smaller problem and the larger problem.

to Jody Forehand, for making it all bearable

Contents

List of Figures

List of Tables

Acknowledgements

I must begin by thanking my wife and then proceed to ask her to forgive me for yet another failing: I am incapable of expressing the depth of my gratitude for her support. She cheerily endured my foulest moods and never let me forget that I was loved. She helped me in countless ways, from bringing me meals at long, lonely nights spent in front of the computer, to reading drafts of papers. She made it all bearable.

Next, I must thank my parents. They helped me move into my first apartment when I came to Berkeley, they proofread every single chapter of my dissertation, and they encouraged me at every step in between. They rejoiced in my successes and comforted me in my tribulations. I could not have asked for more loving or supportive parents.

My advisor, Stuart Russell, has been invaluable. It was at his suggestion that I began studying reinforcement learning. Stuart helped me learn that playing with probabilities could be fun and then continually reminded me of the importance of making my work understandable to people who don't enjoy playing with probabilities. He spent late nights writing papers with me, shared his enthusiasm for my work with his colleagues and introduced me to the academic community. Without his guidance this dissertation might have been an incomprehensible jumble of symbols. I didn't have time to incorporate a few of his suggestions to improve the readability of this document. I hope that he and the gentle reader will forgive me when my prose does not communicate the pictures in mind as effectively as it could or should.

My other readers, Jitendra Malik and Tom Marschak, were wonderfully patient, putting up with what must have seemed like endless delays as I wrestled with typesetting problems, power outages, and the English language itself.

Other members of the Berkeley community, past and present, have been a great help to me: David Andre, John Binder, Huma Dar, Marie desJardins, Doug Edwards, Jeff Forbes, Nir Friedman, Sabine Glesner, Daishi Harada, Othar Hansson, Tim Huang, Keiji Kanazawa, Daphne Koller, Kevin Murphy, Ron Musick, Gary Ogasawara, Nikunj Oza, Vassilis Papavassiliou, Hanna Pasula, Eunice Santos, Prasad Tadepalli, Jonathan Tash, Shlomo Zilberstein, and Geoff Zweig. Of these I am especially thankful to David Andre, Nir Friedman, Tim Huang, Daphne Koller, and Eunice Santos. Their friendship and confidence in me was a great help when my self-confidence waned.

I am proud to belong to an intellectual community that treats hopeful, young graduate students with the same respect as senior researchers. Some of the members of this community who have been particularly helpful and kind to me are Craig Boutilier, Mike Bowling, Tony Cassandra, Tom Dean, Tom Dietterich, Bob Givan, Geoff Gordon, Milos Hauskrecht, Leslie Kaelbling, Sven Koenig, Michael Littman, Sridhar Mahadevan, Doina Precup, Satinder Singh, Rich Sutton, and Manuela Veloso. The scholarship, criticism and advice of these individuals contributed to the quality of this document and to the quality of my life. They helped me stay focused and centered through the process. Any mistakes, of course, are purely mine.

This research also was supported financially through the generosity of the taxpayers: NSF grants IRI-9058427, IRI-9211512, IRI-9634215. ARO grant DAAH04-96-1-0341 and ONR grant N00014-97-1-0941.

At many times in my graduate school experience, I have wondered what the title "Dr." really meant. Was it recognition of a contribution to society, a job requirement, a statement about one's perseverance, or just a meaningless bit of archaic formality? One of the things that always confused me about this is that the title is rarely used by those *inside* of the academic community. It is only now, in the last hours of this experience, that I think I finally understand what the title can and should mean. It is a reminder from people outside the community that those of us who chose to pursue the life of the mind do so at their sufferance. Without the generosity and kindness of strangers, the support of my friends and family and the love of my spouse, my academic pursuits not only would have been impossible, they would have been meaningless.

I leave you with my promise to remember this every time I am addressed.

Ronald Parr, Ph.D.

Chapter 1

Introduction

This chapter is intended to be a mostly non-technical introduction to the topics addressed in this dissertation. It provides an overview of the basic concepts and terminology that will be used and outlines, at a high level, the novel contributions contained in the following chapters.

1.1 Introduction to the Introduction

This dissertation addresses the topic of hierarchical learning and control for stochastic, sequential decision problems. Loosely speaking, a stochastic, sequential decision problem is one that involves decision making in an environment where there is uncertainty. The *sequential* aspect of the decision problem reflects the fact that the immediate cost or benefit of any state of the environment may play only a small part in determining the true value of any state. This permits the appropriate distinctions between states that have an apparent immediate benefit (e.g. being served ice cream) and those that have an apparent benefit but look less desirable when viewed in a larger context (e.g. being served ice cream before a trip to the gallows).

The decision-making problem is framed from the perspective of an *agent* that is situated in an environment. An agent can be a person executing a strategy to solve some task, a robot roaming about an office, or a faceless control program managing an industrial process. The term serves both as a placeholder and as a convenient anthropomorphization.

The aim of algorithms for stochastic, sequential decision making is to develop a conditional plan, or *policy*, that describes a strategy that maximizes the benefit (or minimizes the cost) of acting in a particular environment over an extended period of time. This broad view encompasses countless problems in the intersection between operations research, artificial intelligence, and control theory. Some well-known examples include equipment maintenance (Puterman, 1994), inventory management (Puterman, 1994), autonomous robot control (Mahadevan & Connell, 1992), elevator scheduling (Crites & Barto, 1996), and factory automation (Mahadevan, Marchalleck, Das, & Abhijit, 1997).

The formal framework that is used to describe stochastic, sequential decision problems is the *Markov decision process* (MDP). When a problem description satisfies the requirements of the MDP framework, well-known algorithms can be used to determine an optimal policy. In principle, these algorithms could be used to solve a wide range of problems of great practical and economic importance: autonomous control of automobiles, the design of treatment schedules for medical patients, the planning of space missions or pollution minimization and mitigation. Unfortunately, while many problems can be modeled as MDPs, not all of these problems can be solved easily within this framework. Models that satisfy the requirements of a valid MDP model tend to be extremely large, requiring unreasonable amounts of memory and run-time using standard methods.

This work describes a collection of methods for attacking the complexity of large MDPs through hierarchy and decomposition. It presents novel results on the use of complex, temporally extended actions within the MDP framework and proves the stability and optimality of these methods as a consequence of their relationship to Semi-Markov Decision Processes (SMDPs). These results are first shown to unify a number of approaches developed independently by other authors. This unified view is then used to develop a new class of algorithms called *symbolic* methods, that generalize and extend existing MDP algorithms to handle complex action descriptions and a new class of optimality criteria based upon these descriptions. The new algorithms developed in the beginning of the dissertation provide the basis for a method called Hierarchies of Abstract Machines (HAMs), which allows the incorporation of prior knowledge into the search for good policies. The knowledge contained in a HAM is used to transform a large MDP into a smaller one that makes provably optimal use of the knowledge provided.

This dissertation also attacks large MDPs from another angle, presenting a novel approach to the decomposition of MDPs into independent, or nearly independent subproblems. These subproblems can be solved separately by devising a cache of solutions for each subproblem, and finding the optimal way to combine the cached solutions in a relatively light-weight step. This will produce a global policy that is a bounded distance from the optimal policy. A new polynomial time algorithm for measuring the quality of a set of cached solutions is presented and used as the basis for developing new solution caches. Since this type decomposition may not always be practical if the size of the required solution cache is very large, a method of partially decomposing an MDP into subproblems and intelligently allocating computational resources between the subproblems is also presented. This method uses a novel geometric approach to prove bounds on the distance from the optimal policy based upon a sparse solution cache.

Together, these decomposition methods provide a framework for the transfer of knowledge and experience across problems with similar structures, a long-sought goal of MDP research. The HAM algorithms and the decomposition algorithms are complementary in a very natural way, allowing the incorporation of prior knowledge into the MDP solution process, the use of this knowledge to simplify the MDP, the decomposition of the MDP into separate subproblems based upon the HAM hierarchy, and the transfer of knowledge acquired from solving one problem to new problems.

First, however, the remainder of the introductory section establishes some of the context in which this work resides.

1.2 Decision Making Under Uncertainty

Uncertainty happens. Men, machines, and the universe in general do not always behave in the expected way. There is a temptation to dig in one's heels in the face of uncertainty and insist that all uncertainty is the result of inadequate modeling. This touches on a deep philosophical issue, but also upon an extremely practical one: there is a point of diminishing returns, where the return on effort invested to construct more complicated models no longer merits the investment. At such a point, it may be most efficient to account explicitly for the uncertainty in the environment in order to achieve intelligent behavior. The algorithms discussed here are intended to provide optimal behavior under precisely these conditions.

1.2.1 Some Examples of Uncertainty in Real Problems

A common example used in artificial intelligence is that of a robot moving through some sort of obstacle strewn environment, typically represented as a grid with some squares

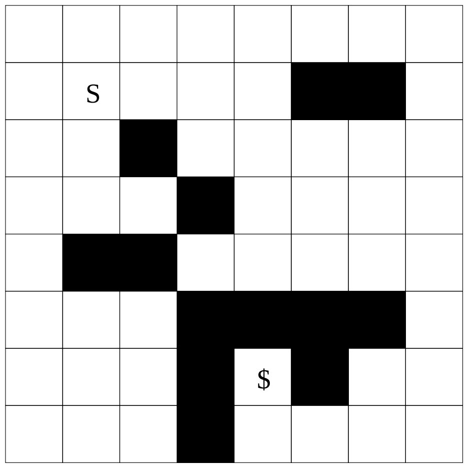

Figure 1.1: A simple navigation MDP. S indicates the starting state, $ indicates a goal state, the attainment of which yields reward.

blocked off and others empty as in Figure 1.1. This representation is meant to model the difficulties faced by a "domestic" robot which might be assigned tasks such as the delivery of parcels from one room to another. Uncertainty is present at many levels even in such simple problems. The robot's motors may not always perform as expected, moving the robot too far, not far enough, or in the wrong direction. The robot's sensors can malfunction, producing erratic or unreliable readings, and errors due to the discrepancy between the robot's representation of the environment and the true dynamics of the environment, can all contribute to make this problem highly non-deterministic from the robot's perspective.

Uncertainty is widespread in many more practical MDPs as well. Inventory management problems (Puterman, 1994) must account for uncertainty in demand levels, while maintenance problems (Puterman, 1994) must deal with the inherent unpredictability of component failures. Elevator scheduling (Crites & Barto, 1996) involves uncertainty in the location and time of the next floor-call, and the number of passengers entering and leaving the elevator at each stop. Industrial plant automation (Mahadevan et al., 1997) involves all of these issues, as parts and personnel must be shuffled from work area to work area in the most efficient manner.

The more ambitious tasks become, the more it appears they are fraught with uncertainty. Controlling an automobile involves massive uncertainty about both the road conditions and the intentions of other drivers. The design of proper treatment regimes for patients involves critical and unpredictable factors such as a patient's tolerance for various drugs. Space missions are highly uncertain due to man's limited ability to determine, in detail, the specific conditions that exist on other planets. These types of uncertainty won't go away: drivers can't take time to call each other on cell phones to communicate their intention to change lanes; patients often need life-saving medicines without delay and in the highest dose they can be expected to tolerate; the planets won't reveal their mysteries unless probes are sent to explore them.

1.2.2 Different Types of Uncertainty

The types of uncertainty faced by agents can be characterized along several dimensions:

- Modeled vs. unmodeled
- Uncertainty in action outcomes
- Uncertainty in perception
- Uncertainty in time

Uncertainty is *modeled* and algorithms using a model are *model-based* if the agent has a representation of the effects of actions, typically a probability distribution over possible outcomes. Uncertainty can be *unmodeled* or *model-free* if no such representation is maintained. Clearly, the presence of a good model can only help in the planning process. In many cases, however, it is impractical to construct a good model *a priori*, and it is necessary to use some form of sampling or learning from the environment to compensate. Most of the algorithms discussed here will work with both modeled and unmodeled uncertainty.

Uncertainty in action outcomes is the best-understood and most tractable form of uncertainty. In this case, the next environment state typically is expressed as some probability distribution over the states of the environment, conditioned on the current environment state and action taken therein. Effectors for simple robots, for example, exhibit this form of uncertainty. Malfunctions or inaccuracy in the robot's control circuitry can cause a joint or even the entire robot to move in a manner that differs from the intended outcome. This uncertainty in the result of the movements can be modeled with a probability distribution. The algorithms discussed here will all work with uncertainty in action outcomes.

Uncertainty in perception or *partial observability* refers to an agent's uncertainty about the current state of the environment. This form of uncertainty is extremely important and extremely common due to the practical impossibility of perceiving accurately all of the relevant features describing a particular situation with 100% accuracy. For example, inexpensive sonars used on robots can have large errors in their distance estimates due to reflections, background noise, or general failures. These can cause significant errors in the robot's own estimation of its position in the environment. General algorithms for this type of partially observable MDP (POMDP)(Lovejoy, 1991) are complicated and can be very inefficient. Moreover, the problem in general is known to be intractable (Papadimitriou & Tsitsiklis, 1987). Some of the algorithms presented here can be useful in attacking special cases of POMDPs, but a discussion of the application of this work to general POMDP models is reserved for future work.

Uncertainty in time refers to uncertainty in the time duration between actions. Problems with this form of uncertainty are called Semi-Markov decision problems or SMDPs (Puterman, 1994). Action uncertainty can arise in many ways. In one view of these situations, the environment is a *discrete event system* (Ramadge & Wonham, 1989), in which the decision-making situations arise as events in the environment and the time between such events is modeled as a probability distribution. A simple example of this is a queuing system, in which the need for a decision, an assignment of a request to queue, arises as a consequence of some exogenous event, the issuance of a request. In another view of SMDPs, actions can be interpreted as complex events that take place over a period of time. For example, a robot action that moves the robot forward until an obstacle is hit is a temporally extended event with uncertainty in both the duration of the action and terminating state of the action. Since robots do not move in perfectly straight lines, the next obstacle the robot hits, the obstacle's distance from the robot's present location, and, thus, the time spent moving will all have uncertainty.

Note that when there is no uncertainty, "classical" AI techniques of search and planning can be used to determine optimal strategies for most domains. These methods can be quite successful when the assumption of zero uncertainty is a valid one. Hierarchical methods for deterministic domains (Tate, 1977) have shown that classical planning methods can scale to solve some very large problems. One interpretation of the work presented here is as an effort to apply the lessons learned from hierarchy and decomposition of deterministic problems to their stochastic counterparts.

1.2.3 Models of planning and acting

An important issue for any planning method is the relationship between the time spent planning and the time spent acting. When there is no overlap between these two, an optimal plan is constructed *off-line* and then used *on-line* without modification. In the example of a robot plan, one could imagine a large, powerful computer producing an optimal plan which is then downloaded to a smaller, mobile computer inside of a robot. The smaller computer is entrusted solely with executing the plan. Off-line planning is the simplest and most easily understood form of planning and is the perspective from which any new approach to stochastic planning should be tested and understood first. All of the algorithms presented in this dissertation can be used for off-line planning.

While off-line planning is the most straightforward form of planning, it also makes the most restrictive assumptions. It assumes that a full model of the environment is available *a priori* and that enough time is available, before the plan must be executed, to construct a full plan covering all states in the environment, even those that are extremely unlikely to occur when the plan is executed. For large problems, this can create an unreasonably difficult planning task.

One approach to loosening the requirements of a full plan is to allow some on-line planning, where the agent executing the plan is allowed to construct or refine a plan as it acts in a modeled environment. This route is desirable in situations where it is impractical to construct a complete plan that covers every possible state an agent might encounter. In such cases it often suffices to use an approach that focuses on the states an agent will most likely encounter while using heuristic or worst-case bounds to handle states for which the agent does not have a full plan. Methods such as Plexus (Dean, Kaelbling, Kirman, & Nicholson, 1995) and real time dynamic programming (RTDP) (Barto, Bradtke, & Singh, 1991) use this approach. This type of on-line planning or plan refinement is not addressed explicitly with the algorithms presented here, but they can be modified in an obvious way to work with Plexus or RTDP.

A more general loosening of the assumptions used in on-line planning allows an agent to start acting in an environment with no initial plan and no model of the environment at all. This regime can be forced upon an agent when it encounters a new environment or any time when a complete environment model is too large or otherwise unwieldy. In such situations, an agent must learn to act optimally the hard way, through experience. The agent can do this by constructing a model as it goes (Sutton, 1990; Moore & Atkeson, 1993; Andre, Friedman, & Parr, 1997) or with no model at all (Watkins, 1989). These methods, which work by incrementally updating estimates of the value of each state, are referred to as *reinforcement learning* (RL), where "reinforcement" refers to the incremental nature of the updates, and the relationship between this type of machine learning and biological models of learning. (See Kaelbling, Littman, and Moore (1996) for a survey.) The approach to temporally extended actions presented here and the HAM-based hierarchical methods are extended to reinforcement learning. The problem decomposition approach presented in Chapter 6 has a less obvious application to RL, but some elements can be carried over to the RL framework.

1.3 The *State* of MDP Solution Methods

This subsection provides a very brief overview of the current state of the art in MDP algorithms and describes some of the limitations of current methods. The three basic categories for for MDP algorithms are:

1. Algorithms that will provably converge to an optimal solution
2. Algorithms that will provably converge to a solution that is some characterizable distance from the optimal solution
3. Algorithms that sometimes converge to something that might not be too far from the optimal solution – maybe.

Since all MDP algorithms are iterative in some sense, convergence means that the algorithm will reach a stable fixed point or, at worst, oscillate in some reasonable area around a fixed point. This is in contrast to divergence, where an algorithm could produce meaningless answers.

1.3.1 Provably Convergent and Optimal Algorithms

Provably convergent and optimal algorithms exist for both off-line and on-line planning. In the off-line case, well-known algorithms such as value iteration, policy iteration or linear programming (see e.g. Puterman (1994)) will produce optimal policies. In the on-line case, Q-learning or Prioritized Sweeping will produce optimal behavior. In their pure form, these algorithms all work by maintaining a big table, with one entry for every state in the environment. They assign a utility value to every state in the environment, working iteratively to successively improve their estimates of the utility of each state. When each state is assigned the correct utility value, the optimal action is just the one that maximizes the expected utility. The run time of these algorithms is linked directly to the number of states and actions in the environment and will grow super-linearly in these factors. (See Littman (1996) for a more thorough discussion of the complexity of solving MDPs.)

This *explicit state* representation provides the strongest convergence and optimality guarantees of any approach to MDPs, but it also is the most restrictive approach. Large problems will require large amounts of memory and even larger amounts of computational resources (or environment experiences in the RL case). Thus, problems with millions of states will not be amenable to this type of direct approach, and such problems are not at all far-fetched: In a robot control program, for example, every possible configuration of the robot and environment would be a distinct state. If the robot is unfortunate enough to have something as simple as a single chess board in its environment, an explicit state representation would require an astronomical number of states, making it quite impractical.

In some cases it will be possible to avoid an explicit representation of the state space while maintaining desirable convergence and optimality properties. Boutilier, Dean, and Goldszmidt (1995) show that it is possible, in some cases, to replace a tabular representation with a compact representation such as a decision tree. Dean and Givan (1997) provide further insight into when a compact representation of a value function can be achieved by assigning the same utility to similar states. In theory, this makes it possible to manipulate vast numbers of states that have the same utility as if they were a single state. These are promising and theoretically appealing methods, but their usefulness depends upon the existence of non-trivial problems with the right kinds of regularity and structure. The density of these types of problems in the set of problems that are of practical interest is not yet known.

Parallelization offers another line of attack for large problems. Papadimitriou and Tsitsiklis (1987) show that this approach does not appear to be very promising in general due to the provable computational difficulty of general MDPs. However, parallel MDP algorithms will converge (Bertsekas & Tsitsiklis, 1989) and some special cases of MDPs that are amenable to parallelism have been explored (Dean & Lin, 1995; Lin, 1997).

1.3.2 Provably Convergent Algorithms

If a suboptimal solution is acceptable, a number of approximate, but provably convergent methods are available. One family of methods works by *state aggregation*, in which states that are "similar" are grouped together and treated as a singled state. This reduces the size of the state space, creating a smaller and easier-to-solve problem. Approximate versions of the Boutilier and Dearden approach and the Dean and Givan approach mentioned above fall under this category when states are grouped together even if they cannot be shown *a priori* to have the same value.

State aggregation can be approached from many different angles, depending upon the notion of similarity that is used to group states together. Boutilier and Dearden use another approach to aggregation (Boutilier & Dearden, 1994) in which states are grouped together based upon their relevance to the agent's expected performance in the environment. If, for example, a robot receives no benefit from playing or winning chess, then a chess board in the environment, and the vast number of states it induces, would all be ignored. In less contrived examples, few features of the environment are totally irrelevant, so the decision to ignore certain features and group states together results in an approximation.

These are just a few of a large number of possible state aggregation methods, each with its own advantages and pitfalls. For off-line solution algorithms, any state aggregation method that produces a valid MDP model will converge to an answer. The difficult part is deciphering how the solution to the aggregated model relates to the original problem. The most conservative aggregation methods group states together only when certain bounds on the relationship between the aggregated states and the original states can be established, providing a roadmap to construct an approximately optimal policy for the original problem based upon the solution to the aggregated problem. Bolder methods group states together based upon expert knowledge or intuition about the similarity of states. These methods provide no performance guarantees, but can work quite well in practice if careful choices are made when the states are aggregated.

There is a significant gap between theory and practice for state aggregation methods. Conservative methods with provable performance guarantees have not yet been demonstrated to be useful for large, practical problems, while bolder methods based on intuition appear to be used either implicitly or explicitly in almost every successful MDP application.

Another important issue with state aggregation is that it may not behave stably in a reinforcement learning context. Off-line state aggregation methods construct and then solve a model that is an approximation of the original problem. On-line methods rely upon the environment to produce a consistent probability distribution for the outcomes of actions. If an agent internally treats two states that are significantly different as the same state, then the agent may not perceive a consistent probability distribution over outcomes, which can lead to oscillations or divergence in the agent's estimate of the utility of states.

Function approximation is closely related to state aggregation, and can be used in provably convergent algorithms in some cases. It replaces the tabular representation for the utilities of states in the explicit state representation with a parameterized function that maps from the states to values. If the table is simply replaced with a smaller table, where several entries in the original table map to a single entry in the smaller table, then function approximation is very similar to state aggregation. It differs significantly from state aggregation when more advanced function approximation methods are used. For example, a neural network could be used to represent the utility of states, making it possible to represent a utility function for a large number of states implicitly with a much smaller number of network parameters. In this way, function approximation is much more versatile than state aggregation is since it permits smoothing and generalization across similar states, rather than simply forcing them to have the same value.

Gordon (1995) establishes the convergence of a class of function approximation methods for MDPs. This guarantees that for a fairly restricted class of approximation methods, a stable utility function that is within some well-characterized distance from the true one will result. Baird (1995) establishes convergence for more general classes of approximators, such as neural networks, but this method is awkward to use in practice, the convergence may be extremely slow and the relationship between the resulting utility function and the true utility function is extremely difficult to characterize.

1.3.3 Algorithms that Might Converge to Something — Maybe

Some of the most ambitious function approximation methods use approximators like neural networks in a reinforcement learning context and in a manner that is not guaranteed to converge at all. In fact, there are well-known examples where such combinations will diverge (Boyan & Moore, 1995). Still, there are some intriguing examples of success with this approach: in Backgammon (Tesauro, 1989), in job shop scheduling (Zhang & Dietterich, 1995), and in elevator scheduling (Crites & Barto, 1996).

Getting function approximators to work with reinforcement learning is a delicate art, requiring careful adjustment of the parameters to the neural network and making the discovery of the right set of parameters a tedious process of trial and error. The draw of methods such as these, which lack performance guarantees or provable properties of any kind, underscores the gap between theory and practice. Too many problems of practical interest are simply too big for the formally certified tools that are available.

1.4 Contributions of this Dissertation

The outlook for algorithms that have attractive formal properties *and* that can be used to solve large MDPs is not all depressing. Beginning with the concept of temporal abstraction, this dissertation develops new methods involving hierarchy and decomposition that provide tools with provable optimality and convergence properties for solving large MDPs. These tools provide a principled basis for the transfer of knowledge and experience across problems and will help narrow the gap between theory and practice.

1.4.1 Temporal Abstraction

Temporal abstraction refers to the use of "high-level" actions, the execution of which can take varying amounts of time. This is in contrast to "primitive" or "low-level" actions which are used in the standard MDP framework and which are assumed to take a uniform amount of time. Temporal abstraction permits a richer notion of actions, such as the action of moving to the end of a hallway, driving to one's house, or scratching until an itch goes away. These complex actions take variable amounts of time that cannot be determined *a priori*.

Many on-line and off-line MDP algorithms have used implicitly or explicitly some form of temporal abstraction (Forestier & Varaiya, 1978; Lin, 1993; Sutton, 1995; Dean & Lin, 1995). Chapter 3 unifies these varied approaches by demonstrating that a temporally abstract action is equivalent to the transformation of a policy defined over a region of an MDP into an action in a related Semi-Markov Decision Process (SMDP). This connection between temporal abstraction and SMDPs permits the establishment of convergence and optimality results for temporal abstraction algorithms as a consequence of the convergence and optimality results for standard SMDP algorithms. Chapter 3 also proves the convergence of a variant of Q-learning that has been modified to work with SMDPs (Bradtke & Duff, 1995), defines a notion of equivalence between SMDPs, and proves basic theorems about the soundness of operations for transforming temporally abstract MDP actions into SMDP actions.

Chapter 4 presents a new class of algorithms designed to efficiently execute the transformations of Chapter 3 in direct, indirect, on-line and off-line fashions. This chapter also demonstrates the connection between temporal abstraction and asynchronous dynamic programming, showing that a temporal abstraction is equivalent to a specific ordering of asynchronous dynamic programming operators. The new understanding of temporal abstraction that is developed in these chapters allows the description of a new set of optimality criteria for MDPs. These criteria can be used to generate temporally abstract actions according to high level criteria such as the maximization of the probability of reaching a particular group of states. A new algorithm is presented for such high-level criteria that will simultaneously discover the optimizing policy and the equivalent SMDP action description.

1.4.2 Knowledge Transfer and Reuse

Many problems can seem large or difficult when viewed from the ground up, but when they are viewed in the context of previous experience with similar problems, or when a knowledgeable expert is available as a guide, these problems suddenly become much easier. With a few exceptions (Thrun & Schwartz, 1995) MDP algorithms have not employed sophisticated or principled means for reusing and transferring knowledge. Chapter 5 introduces the HAM language for transferring knowledge from human to machine and machine to machine. This language takes the form of a partially specified, hierarchical, finite state controller that can be interpreted as a set of constraints on the solutions that are considered for a new MDP. This view of knowledge as a constraint may seem peculiar at first, but it is consistent with the way people approach problems. For example, when planning a trip from one's house the grocery store, one does not replan the detailed control decisions needed to operate a motor vehicle or try to invent a hovercraft. Basic skills and assumptions are taken for granted and solutions that involve discovering a new mode of transportation are implicitly ruled out. HAMs exploit this basic idea and Chapter 5 shows how the structure of a HAM is used to transform large MDPs into simpler SMDPs in such a way that the solution to the simpler SMDP makes provably optimal use of the knowledge contained in the HAM. This transformation uses the concepts of temporal abstraction developed in the previous chapters by using temporally abstract actions to implement machine subroutine calls. The effectiveness of this approach is demonstrated on a large MDP.

Chapter 6 takes a different and complementary approach to the reuse of knowledge. While HAMs transfer procedural knowledge, the decomposition algorithms in this chapter exploit structural similarities between different problems to transfer experience from one problem to another. These algorithms exploit the idea of a stored *cache* of policies that can be used for recurring problem substructures. When a new problem is encountered, a solution from the policy cache can be employed, or a simple linear program can be constructed to *prove* bounds on the cost of using a cached solution instead of constructing a new one. For example, a robot might have a cache of policies for moving around the first floor of a building. When the robot encounters a new problem that specifies a goal of reaching the basement, the robot can plug in a first-floor policy from its cache or it can construct an entirely new policy for the first floor, taking the new goal into account. Chapter 6 provides insight into and a *quantifiable* measurement of the tradeoffs between reusing transferred knowledge and discovering new knowledge.

1.4.3 Hierarchy and decomposition

Hierarchical representations are crucial to the efficient representation of knowledge. Knowing how to turn a steering wheel or operate a gear shift, by themselves, are fairly useless. Knowing that these things can come together in service of the task of driving a vehicle and that driving a vehicle is a means of transportation are necessary bits of conceptual glue that make the difference between an unstructured heap of knowledge and knowledge that is useful to accomplish real tasks. HAMs provide a hierarchical language that permits tasks to be described as loose combinations of lower-level tasks. This separates high-level decisions from low-level decisions and provides guidance as to the appropriate time to make low level decisions. HAMs can convey the obvious but crucial information, that one should decide *how* to go to the grocery store before even considering the mechanics of turning a steering wheel or moving a gear shift. They are an efficient and compact means of transferring knowledge.

HAMs also provide a natural decomposition of a task into subproblems. By creating distinctions between different types of activities, they create a hierarchy of task-related subproblems. For example, choosing a mode of transportation constitutes one high-level problem. Choosing optimal bus transfers or automobile routes constitute different subproblems that are activated depending upon the high-level transportation mode choice. Since MDP subproblems tend not to be completely independent except in special cases (Singh, 1992; Lin, 1997) some care must be used when problems are decomposed. Chapter 6 presents a new approach to the decomposition of MDPs. This approach applies to the special, but frequently occurring type of MDPs that can be divided into subproblems where the number of states connecting the subproblems is small. These problems can, in principle, be divided into completely independent subproblems, where each subproblem is solved by constructing a *cache* of solutions. Each solution in the cache is interpreted as a temporally abstract action in the sense of Chapter 3 and the optimal combination of the cached solution is determined efficiently. The relationship between the combined solution and globally optimal solution can be determined *a priori* with a new algorithm that will produce solution caches that guarantee optimality within pre-specified bounds. This relationship can also be determined on-the-fly by using the same techniques used for knowledge transfer. Thus, a fairly sparse cache of policies can be constructed for each subproblem, and bounds can be proven as the subproblem solutions are combined. This quantifies the cost of using a sparse solution cache versus the cost of augmenting the cache with more solutions. The effectiveness of these algorithms is demonstrated with some examples.

1.4.4 The Big Picture

This dissertation presents powerful new algorithms for solving large MDPs. These algorithms assume an explicit state representation of the MDP, but aim to avoid the difficulties of explicit representations by never solving a large problem in its entirety. First, knowledge in the form of a HAM can be used to produce a smaller SMDP, then the structure of the HAM can be used to hierarchically decompose the SMDP into subproblems that are solved (mostly) independently. This divide-and-conquer approach has the potential for a dramatic reduction in the computational effort required to solve large MDPs. The advantages of these algorithms are that they provide a principled means of incorporating prior knowledge, they have well-characterized convergence and optimality properties, and they use novel error-bounding methods to minimize the interaction between subproblems.

State aggregation and function approximation are largely orthogonal to the approach advocated here, but some possible synergies are discussed briefly in Chapter 7 along with the issue of partial observability. Several extensions to the HAM language are sketched in this chapter as well, providing a few examples of the many interesting areas of future research that can build on the ideas in this dissertation.

The ideas developed here point in a direction that is significantly different from that taken by much of the applied research in this area today. Those seeking to solve large MDPs today typically ask themselves two questions:

1. What is a relatively well-behaved function approximation or state aggregation method for this problem?
2. How do I select the inputs to this function approximation method?

These questions are very difficult to answer and the answers tend not to be very general. The resulting solutions are sometimes quite impressive, but are often quite frustrating. The tools presented in this dissertation suggest some different questions:

1. How can I bring knowledge and experience to bear on this problem?
2. How does this problem decompose?

For many domains, the answers to these questions are both more readily available and more easily transferred to new problems. The algorithms in this dissertation provide the means for exploiting these answers in a principled way, making the solution of large MDPs more of a science and less of an art.

Chapter 2

Markov Decision Processes

This chapter introduces the Markov Decision Process (MDP) formalism. In this formalism, an environment is divided into states. A set of actions induce stochastic state transitions in the environment and determine the evolution of the process. The aim is to devise a conditional plan, or *policy* that will maximize the expected benefit (or minimize the expected cost) of interacting with the environment. Several standard algorithms rooted in operations research will find an optimal policy for an MDP as a mapping from states to actions. These algorithms are presented along with common notation and terminology that are used throughout the dissertation. On-line variants, called *Reinforcement Learning* that are more closely tied to Artificial Intelligence and that learn the optimal policy through experience with a previously unknown domain are also presented.

The standard algorithms for MDPs are efficient in the number of states and actions in the MDP. However, the formal requirements of the MDP framework, specifically the Markov property, force representations of problems that use a very large number of states. This chapter explains, in a more mathematical sense, the difficulties with large state spaces that were outlined in the previous chapter and then presents a more technical discussion of some common methods for dealing with large state spaces.

2.1 Basic Terminology

Formally, an MDP is a 4-tuple, $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$. $\mathcal{S}$ is a set of states. In general, $\mathcal{S}$ may be of infinite cardinality, but for the purposes here it is assumed to be finite. A set of actions, $\mathcal{A}$, defines the actions that are possible or under consideration in the environment.

It may be the case that only a subset of $\mathcal{A}$ is possible in any given state, so $\mathcal{A}_s$ is used to refer to the set of actions possible in $s \in \mathcal{S}$. $\mathcal{A}$ will generally be finite although it will sometimes be useful to consider infinite cardinality $\mathcal{A}$, usually when an action corresponds to selecting some parameter in a continuous range. $\mathcal{T}$ is the transition model for the system and is a mapping from $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$ into probabilities in $[0,1]$. $\mathcal{T}(s,a,s') = P(s'|s,a)$, the probability that the next state will be s' when action a is taken in state s.

$\mathcal{R}$ is a reward function that maps from $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$ to real-valued rewards. $\mathcal{R}(s,a,s')$ is interpreted as the mean of a stochastic function that returns rewards upon transitions from state s to state s' under action a. $\mathcal{R}(s,a,'s)$ must be finite for all s, a, and s'. From the perspective of determining an optimal policy for an MDP, there is no distinction between a reward function defined on $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$ and one defined on $\mathcal{S} \times \mathcal{A}$, as the destination state can be averaged out with no effect on planning:

$$R(s,a) = E_{s'}[R(s,a,s')] = \sum_{s'} T(s,a,s')\mathcal{R}(s,a,s').$$

The two representations are used interchangeably here. It is sometimes more convenient to associate rewards with states rather than in either of these more general forms.

The states and transitions functions must be defined in such a way that they satisfy the Markov property: the probability of the next state must be a function of the current state and action only, not any of the previous states. More formally, for any action a and string of states $s_1 \ldots s_t$, $T(s,a,s') = P(S_{t+1} = s|a, s_t \ldots s_1) = P(s_{t+1} = s|a, s_t)$, where s_t is a random variable for the state of the environment at time t.[1] The Markov property makes it possible to write $P(s_x|s_y, a)$ with no references to time, as the probability that the next state will be s_x given that the current state is s_y and action a is taken. For discrete $\mathcal{S}$ and discrete $\mathcal{A}$, $\mathcal{T}$ can be thought of as a set of transition matrices representing the conditional probability of the next state given the current state, and each $a \in \mathcal{A}$ can be thought of as selecting an element from this set.

A policy, π, for an MDP is a scheme for assigning actions in $\mathcal{A}$ to states in $\mathcal{S}$. Policies may be *stationary*, which means that they make a simple mapping from states to actions, or they may be *non-stationary* which means that the action assigned to a particular state may change depending upon some other factor, like the number of steps the agent

[1]Authors in this field use a subscripted s somewhat inconsistently. Sometimes s_i is a random variable indicating the state at time i and sometimes it is a label for i^{th} element of $\mathcal{S}$. The latter usage is the default here, and whenever the former is intended, the reader will be warned explicitly that s_i is a random variable.

has taken in the environment, or some function of the agent's past experiences in the environment. A family of policies that optimize the agent's behavior with respect to some optimality criterion is referred to as Π^*. Algorithms for MDPs typically search for some optimal $\pi^* \in \Pi^*$ or an approximation thereof.

A value function, V, is a mapping from elements of $\mathcal{S}$ to real values. A value function is usually thought of as representing the utility of a state or the "cost-to-go" for a state, which is some measure of the cost (or reward) for continuing in the environment from that state. If a policy π assigns to each state $s \in S$ the action that maximizes the expected value of s given the values assigned to s and states reachable from s in one step by some value function V, then π is said to be *greedy* with respect to V. These concepts will become more concrete in the following section where optimality criteria are defined.

2.2 Optimality Criteria

For there to exist an optimal policy, there must exist an optimality criterion that places some kind of ordering on different policies. A typical optimality criterion is the expected, discounted sum of rewards received by the agent:

$$\sum_{t=0}^{N} \beta^t \mathcal{R}(s_t, \pi(s_t))$$

where s_t is a random variable indicating the state of the environment at time t, and β is a *discount factor* in $[0 \ldots 1)$. The discount factor reflects the value of time and indicates that one unit of utility one time step in the future is worth β units at the present. In economic terms, if the inflation rate is i, then $\beta = 1 - i$. Note that since rewards are finite, the sum above must also be finite even if $N = \infty$. In some cases $\beta = 1$ can be allowed if it does not cause the summation to be unbounded.

Expected discounted total reward is the focus in this dissertation. However, other optimality criteria are possible. In many problems where there is no reason to consider inflation, the average reward per time step is a more natural criterion (Mahadevan, 1996).

The size of N, the last time step, determines the form of the target solution. When N is finite, the aim is to find a *finite-horizon* policy. The optimal finite horizon policy may be non-stationary in the sense that different actions may be selected for the same state at different time steps. In problems with hard time deadlines, finite horizon policies are a natural choice. In problems without hard deadlines, $N = \infty$, and an *infinite-horizon* policy is desired. It is a theorem (Blackwell, 1962) that the optimal infinite horizon policy is stationary. This is a very useful property: if it were non-stationary it could require an infinite amount of storage space to represent the optimal policy for every possible state and time step. Such a policy would not be computable.

Unless stated otherwise, infinite-horizon policies that maximize expected, discounted reward will be the focus of what follows.

2.3 Value functions and the Bellman Equation

MDP algorithms work by assigning values to states and manipulating these values in a manner that reveals the optimal policy. Any policy, π, defines a value function, V_π over the states in the model such that:

$$V_\pi(s) = \mathcal{R}(s, \pi(s)) + \sum_{t=1}^{\infty} \beta^t \mathcal{R}(s_t, \pi(s_t))$$

where the s_t are random variables for states encountered t steps in the future.

The Bellman equation (Bellman, 1957) from dynamic programming establishes a relationship between $V_\pi(s)$ and and V_π for other states in the model:

$$V_\pi(s) = \mathcal{R}(s, \pi(s)) + \beta \sum_{s'} \mathcal{T}(s, a, s') V_\pi(s'). \qquad (2.1)$$

This says that the value of state s under policy π is the immediate reward received in state s plus the expected value of the succeeding state.

There exists a unique, optimal value function, V^*, (Blackwell, 1962) resulting from any optimal policy, π^*, such that:

$$V^*(s) = \mathcal{R}(s, \pi^*(s)) + \beta \sum_{s'} \mathcal{T}(s, a, s') V^*(s')$$

Since the optimal policy assigns the best action to every state:

$$V^*(s) = \max_a [\mathcal{R}(s, a) + \beta \sum_{s'} \mathcal{T}(s, a, s') V^*(s')] \qquad (2.2)$$

Notation adapted from Bertsekas and Tsitsiklis (1989)[2] provides a convenient way to express the right-hand-sides of these frequently occurring forms of the Bellman equation.

[2]The *style* of notation is adopted, but the symbols are different. In Bertsekas and Tsitsiklis (1989) J is a value function a T is an operator.

J_a is an operator that is applied to a value function and that determines the right hand side of the Bellman equation for state s under action a:

$$V_\pi(s) = J_\pi(s)V_\pi$$

When there is no policy subscript, J applies a maximization over actions as in equation 2.2 for V^* above. This yields:

$$V^*(s) = J_{\pi^*}(s)V^* = J(s)V^*.$$

When there is no state argument to J, it operates on vectors, i.e. entire value functions, and the subscript indicates a policy that is applied to the entire state space:

$$V_\pi = J_\pi V_\pi$$

and

$$V^* = J_{\pi^*}V^* = JV^*.$$

2.4 Value Determination

Value determination is the process of determining the expected, discounted sum of rewards that will be accrued from each state under a particular policy. The Bellman equation establishes a relationship that must hold between all states in the model for a particular policy. It also provides a means of computing this value function. For a fixed policy, π, there are two main off-line methods: successive approximation and linear equation solving. For on-line value determination, the method of temporal differences (Sutton, 1988) determines V_π through experience with the environment.

2.4.1 Successive approximation

The method of successive approximation, which is an instance of Gaussian iteration, starts with an arbitrary value function, V^0, and treats the right hand side of the Bellman equation as an assignment. Specifically,

$$V^i \leftarrow J_\pi V^{i-1},$$

With each application of J_π, V^i will get closer to V_π. It is a theorem (Blackwell, 1962) that:

$$V^\infty = J_\pi^\infty V^0 = V_\pi$$

where J_π^∞ expresses an infinite number of applications of the J_π operator. This is true because J is a *contraction mapping* in maximum norm. The maximum norm, $\|\cdot\|$, between V^j and V^k is expressed as:

$$\|V^j - V^k\| = \max_s |V^j(s) - V^k(s)|$$

and the contraction property of J ensures that

$$\|V^j - V^k\| = \beta \|J_\pi V^j - J_\pi V^k\|$$

where β is the discount rate for the model. This defines a fixed point for the J operator and defines a rate, β, at which successive approximation approaches the fixed point. Note that this does not provide a closed form solution for V_π, but provides a means of getting arbitrarily close to V_π at an exponentially fast rate.

2.4.2 Asynchronous successive approximation

Standard successive approximation works by updating the entire value function in one step: $V^i \leftarrow J_\pi V^{i-1}$. A more general form of this operation is possible by considering asynchronous updates in the spirit of Gauss-Seidel iteration. At each step, some $G \subseteq \mathcal{S}$ is chosen and for each s in G,

$$V^i(s) \leftarrow \begin{cases} J_\pi(s)V^{i-1} & \text{if } s \in G \\ V^{i-1}(s) & \text{if } s \notin G \end{cases}$$

If the strategy for selecting G at each stage ensures that every s is updated infinitely often, then V^∞ will contract to V^π (Bertsekas & Tsitsiklis, 1989). The contraction rate will depend upon β and the frequency at which the least frequently updated state is updated. Asynchronous value determination is significant both because it provides insight into on-line learning methods such as TD (Section 2.4.4) and because it provides an avenue for the parallelization of value determination. For example, different groups of states could be parceled out to different processors. The processor assigned to s and the processor assigned to some s' reachable from s may not be the same, forcing the processors to communicate with each other about their current estimate of V_π. Since the bandwidth between the processors will be finite, communications may be subject to delays and V_π may not be synchronized or consistent across all of the processors. The convergence of asynchronous value determination ensures that as long as the processors eventually convey updated value information to each other, and can do so infinitely often, the value functions will converge.

2.4.3 Linear equation solving

A second method for value determination works by solving a system of linear equations. This approach uses the observation that the Bellman equation for policy π is linear in the values of the other states. By writing out the Bellman equation for each $s_1 \dots s_n \in \mathcal{S}$, this produces a system of linear equations:

$$\begin{aligned} V_\pi(s_1) &= J_\pi(s_1)V_\pi \\ &\vdots \\ V_\pi(s_n) &= J_\pi(s_n)V_\pi \end{aligned}$$

where the $V_\pi(s_i), 1 \leq i \leq n$ are the free variables in the system. In matrix form, this system is expressed as:

$$\left(\begin{array}{cccc|c} 1-\beta T(s_1,\pi(s_1),s_1) & \beta T(s_1,\pi(s_1),s_2) & \beta T(s_1,\pi(s_1),s_3) & \dots & \mathcal{R}(s,\pi(s_1)) \\ \beta T(s_2,\pi(s_2),s_1) & 1-\beta T(s_2,\pi(s_2),s_2) & \beta T(s_2,\pi(s_2),s_3) & \dots & \mathcal{R}(s,\pi(s_2)) \\ \beta T(s_3,\pi(s_3),s_1) & \beta T(s_3,\pi(s_3),s_2) & 1-\beta T(s_3,\pi(s_3),s_3) & \dots & \mathcal{R}(s,\pi(s_3)) \\ \vdots & \vdots & \vdots & \ddots & \vdots \end{array}\right)$$

The solution to this system of equations produces V_π. The system can be solved using Gaussian Elimination in $O(n^3)$ time, where n is the number of variables in the system. This can be driven closer to $O(n^2)$ with methods that exploit sparseness in the matrix.

2.4.4 Temporal differences

Temporal difference (TD) learning (Sutton, 1988) is an on-line method that assumes no *a priori* knowledge of $\mathcal{T}$, the transition function or, $\mathcal{R}$, the reward function. Instead, these factors are accounted for implicitly by sampling from the environment and applying an incremental variant of the J operator. When the agent executes $\pi(s)$, observes a transition from state s to state s' and receives reward r, TD adjusts V(s) as follows:

$$V^i(s) \leftarrow \alpha_i(s)[r + V^{i-1}(s') - V^{i-1}(s)]$$

where $0 \leq \alpha_i(s) \leq 1$ is a learning rate, that is indexed by state and that decays over time, and $r + V^{i-1}(s')$ is the sampled right-hand-side of the Bellman equation. For $t \neq s$,

$$V^i(t) \leftarrow V^{i-1}(t)$$

TD adjusts the value of state s in the direction of the right-hand-side of the Bellman equation with a step size of $\alpha_i(s)$. Adopting the notation from the off-line algorithms, an operator, $\mathcal{J}_\pi(s)$, applies the TD operator to a value function at state s, which makes TD expressible as

$$V^i(s) \leftarrow \mathcal{J}_\pi(s)V^{i-1}.$$

It is a theorem (Sutton, 1988) that with probability 1, V^i will converge to V_π when the $\mathcal{J}_\pi$ TD operator is applied to the infinite sequence of states, $s_0 \ldots s_\infty$, that results when policy π is applied in the environment and when $\alpha(s)$ is decayed at a suitable rate, i.e. when $\sum_i \alpha_i(s) = \infty$ and $\sum_i \alpha_i^2(s) < \infty$.

The intuition behind this result is that the $\mathcal{J}_\pi$ operator is an incremental, sampled version of the asynchronous J_π operator and that it is a contraction in expectation. This describes just one specific variant of TD called TD(0). TD can be generalized to propagate information across trajectories of states, not just across pairs of states and this generalization, which is called TD(λ), also converges.

2.5 Algorithms that determine π^*

The following algorithms extend the tools and intuitions developed for determining the value of a single policy to determine the optimal policy, π^*.

2.5.1 Value Iteration

Value iteration is very similar to value determination. It starts with an arbitrary value function, V^0, but instead of applying J_π for a particular policy, it applies the general J operator, which contains a max over all actions:

$$V^i \leftarrow JV^{i-1},$$

and it is a theorem that

$$V^\infty = J^\infty V^0 = V^*.$$

As in the value determination case, this result follows from the contraction property of the J operator, which means that V^i will approach V^* at contraction rate β. The optimal policy can be extracted from V^* by using the operator $\mathsf{greedy}(V)$, which maps from value functions to policies:

$$\mathsf{greedy}(V, s) = \arg\max_a J_a(s)V$$

greedy is so named because it greedily chooses the action with the highest apparent value in each state.

Since the contraction property of J ensures only that V^i will get arbitrarily close to V^*, it is useful to have some means of determining the value of $\pi_i = \mathsf{greedy}(V^i)$. Williams and Baird (1993) provide a bound which is based upon the *Bellman error* or *Bellman residual*, which is defined for a state s and value function V as:

$$BE(V, s) = J(s)V - V(s).$$

When the s argument is dropped, BE refers to the maximum norm of the Bellman error:

$$BE(V) = \|JV - V\|$$

Baird and Williams show that for $\pi_{i+1} = \mathsf{greedy}(V^i)$,

$$V^{\pi_{i+1}}(s) \geq V^*(s) - \frac{2BE(V^i)}{1-\beta} \tag{2.3}$$

which has the obvious implication that the value of the greedy policy for V^i will move closer to V^* as V^i contracts towards V^*. However, it does not guarantee that this progress will be monotonic as the maximum Bellman error may increase or oscillate slightly initially, even as V^i makes progress towards V^*.

2.5.2 Asynchronous Value Iteration

As with value determination, value iteration can be generalized to the case where not all states are updated at once. At each step, some $G \subseteq \mathcal{S}$ is chosen and for each s in G,

$$V^i(s) \leftarrow \begin{cases} JV^{i-1} & \text{if } s \in G \\ V^{i-1} & \text{if } s \notin G \end{cases}$$

This will contract to V^* as long as a scheme for selecting G ensures that every state will be updated infinitely often. The contraction rate will depend upon the discount factor and the frequency with which the least frequently updated state is updated. Asynchronous value iteration provides the basis for establishing the convergence of Q-learning (Section 2.5.6) and for the parallelization of value iteration.

2.5.3 Policy Iteration

While value iteration works by incrementally updating the value function and policy together, policy iteration works iteratively by first freezing the policy, determining the value function for the policy, determining a policy that improves upon the old one based upon this value function and then repeating the process until the policy cannot be improved upon. This can be expressed in pseudo-code as:

```
procedure policy-iteration
  $\pi_0 \leftarrow$ an arbitrary policy
  $j \leftarrow 0$
  continue $\leftarrow$ true
  While continue
    Compute $V_{\pi_j}$
    $\pi_{j+1} \leftarrow$ greedy($V_{\pi_j}$)
    if $\pi_j = \pi_{j+1}$ then
      continue $\leftarrow$ false
    else
    $j \leftarrow j + 1$
  return($\pi_j$)
```

Policy iteration converges at least as quickly as value iteration (Puterman, 1994; Littman, Cassandra, & Kaelbling, 1996) and must find the optimal policy in a finite number of iterations since the policy improves at every step and there is a finite number of policies. It is generally believed to produce optimal policies faster than value iteration does in practice. The greedy policy bound (equation 2.3) can be used to bound the difference in expected, discounted reward between following the optimal policy and following policy π_j in terms of the maximum Bellman error of policy $j-1$. A result from Williams and Baird (1993) also bound the distance between V^{π_j} and V^* in terms of the maximum Bellman error of the current policy:

$$V_{\pi_j}(s) \geq V^*(s) - \frac{BE(V^i)}{1-\beta}$$

Although policy iteration is considered to be the faster route to π^* in practice, it is by no means computationally inexpensive. Each phase must solve a system of linear equations with one equation for each state. The only bound on the number of bound on the number of policies produced before discovering the optimal policy is (Littman et al., 1996):

$$\frac{1}{1-\beta} log \frac{1}{1-\beta}.$$

2.5.4 Asynchronous Policy Iteration

Asynchronous policy iteration combines elements of asynchronous value determination, asynchronous value iteration and policy iteration. In its most general form (Singh & Gullapalli, 1993; Bertsekas & Tsitsiklis, 1996), asynchronous policy iteration operates on subsets $G^1 \subseteq \mathcal{S}$ and $G^2 \subseteq \mathcal{S}$, performing:

$$V^i(s) \leftarrow \begin{cases} J_{\pi^{i-1}} V^{i-1} & \text{if } s \in G^1 \\ V^{i-1} & \text{if } s \notin G^1 \end{cases}$$

and

$$\pi^i(s) \leftarrow \begin{cases} \mathsf{greedy} V^{i-1}(s) & \text{if } s \in G^2 \\ \pi^{i-1}(s) & \text{if } s \notin G^2 \end{cases}$$

with no restrictions on the relationship between G^1 and G^2. A necessary condition for convergence is that the scheme for selecting G^1 and G^2, as in the other asynchronous algorithms, guarantees that both the policy and the value are updated infinitely often. Ensuring convergence for asynchronous policy iteration requires some additional provisions as well. Singh and Gullapalli require that value updates be “single-sided” which means that the assignment, $V^i(s) \leftarrow J_{\pi_{i-1}} V^{i-1}$ is replaced with

$$V^i(s) \leftarrow \max(V_{i-1}, J_{\pi_{i-1}} V^{i-1}).$$

They also require $V^0 \leq V^*$. Bertsekas and Tsitsiklis require only $J_{\pi_0} V^0 \geq V^0$.

If asynchronous value iteration is initialized with a phase that performs a single synchronous value iteration or policy iteration step, these conditions are satisfied easily. This extremely general algorithm is intriguing because it encompasses a large variety of algorithms. Policy iteration, value determination, value iteration, asynchronous value determination, asynchronous value iteration, and modified policy iteration (Puterman, 1994) are all special cases of asynchronous policy iteration.

2.5.5 Linear Programming

The optimal value function, V^*, can be expressed as the solution to a linear program. The right-hand-side of the Bellman equation is treated as a constraint:

Minimize:

$$\sum_s V^*(s)$$

Subject to:

$$V^*(s) \geq J_a V^* \quad \forall s, a$$

It may seem counterintuitive that this minimization would produce V^*, which is the highest attainable value function. Note, however, that any value function less than the true V^* would violate one of the constraints. Any value function greater than the true V^* could be changed to improve the objective function by reducing the value of some state.

The linear programming formulation for the value function also provides a quick means of extracting the optimal policy. Each constraint in the system corresponds to an action selection for a state. A non-zero slack variable for an inequality in the linear program indicates that the corresponding action is maximizing. This also means that each point in the simplex corresponds to a specific policy for the MDP. This makes iterative linear programming methods that work by moving between simplex vertices instances of policy iteration.

In spite of the elegance of this formulation, linear programming does not appear to be the best approach to solving large MDPs. Anecdotal evidence suggests that policy iteration performs better for most problems. This may be because policy iteration uses a more powerful method of selecting new policies (Littman, Dean, & Kaelbling, 1995) than the method used by most linear programming methods to select new simplex vertices. However, there is not yet a definitive analysis of this issue.

2.5.6 Q-learning

One would think that an incremental form of asynchronous value iteration would converge to V^*, just as TD(0), which is the incremental form of asynchronous value determination converges to V_π. This approach would require that the agent also learn a model of the environment that it could use to determine the greedy action based upon the current value function. While intuitively appealing, this scheme is not guaranteed to converge.

Q-learning (Watkins, 1989) is a variation on this basic idea which guarantees convergence. It maintains an indexed version of a value function called a Q function, where $Q(s, a)$ represents the value of taking action a in state s. $Q_\pi(s, a)$ represents the value of taking action a in state s and continuing under policy π afterwards. $Q^*(s, a)$ represents the value of taking action a in state s and acting optimally thereafter. Clearly, $V_\pi(s) = Q_\pi(s, \pi(s))$, $V^*(s) = \max_a Q^*(s,a)$, and $\pi^*(s) = \arg\max Q^*(s,a)$. Upon a transition from state s to state s' under action a and with reward r, a Q-learning agent performs the following TD(0) style update:

$$Q^i(s,a) \leftarrow Q^{i-1}(s,a) + \alpha^i(s,a)[\beta V^{i-1}(s') + r - Q^{i-1}(s,a)]$$

where $\alpha^i(s,a)$ is a learning rate that is indexed by state and action and that decays as in the TD(0) case. For $t \neq s$ or $a' \neq a$:

$$Q^i(t,a') \leftarrow Q^{i-1}(t,a')$$

It is a theorem that Q^∞ will converge to the optimal Q^* values with probability 1 (Jaakkola, Jordan, & Singh, 1994) as long as $\sum_i \alpha^i(s) = \infty$ and $\sum_i \alpha_i^2(s) < \infty$. This has the peculiar side effect of forcing the agent to take actions that are not greedy, i.e. appear suboptimal. In fact the agent must try every action in every state infinitely often to ensure that the summation over α is infinite. Thus, in order to obtain an optimal policy, the agent must act suboptimally — forever. In practice it is often best to choose the greedy action most of the time and suboptimal actions a smaller percentage of the time, with this percentage decreasing as i increases. A common strategy is to select action a with probability determined by the Boltzmann distribution with temperature T:

$$\frac{e^{Q(s,a)/T}}{\sum_{a'} e^{Q(s,a')/T}}.$$

This assigns probability 1 to the maximizing action as $T \to 0$, but will approach a uniform distribution for large T. Typically T is initialized to some large value and decayed over time.

2.6 Abstraction and Approximation in MDPs

The algorithms described in the previous section all assume an explicit state representation of the MDP — the value function is a vector with one element for each state. This representation is impractical for very large problems, both in terms of the space required to store the value function and in terms of the run time required to produce an optimal policy. Standard methods for avoiding this difficulty still use the same basic algorithms, but employ some form of abstraction or value function approximation.

2.6.1 State Abstraction

State abstraction or state aggregation refers to a general class of methods where a single state is used to represent a large group of states. This produces a reduced state space, which is easier to solve using the standard MDP algorithms discussed in this chapter. Note that state abstraction is common in traditional planning, where only the relevant features of any state, typically some set of state variables, are used to represent a large class of states where the other variables have "don't care" values.

Difficult questions for state abstraction in MDPs arise because of the complex value relationships that can exist between seemingly unrelated states. Where traditional planning assumes a deterministic model with a goal of achievement, the stochasticity of MDPs and optimality criteria for MDPs can easily induce an optimal value function that assigns different values to every state and an optimal policy that implies some form of utility relationship between any two states. In some cases, it is possible to show that MDP states can be aggregated without any effect on solution quality (Lin, 1997; Dean & Givan, 1997), but state abstraction generally involves a tradeoff between optimality and compactness. Difficult issues that must be resolved are

1. The manner in which a transition model and reward function for the abstract model are derived from the original model.

2. The relationship between the solution to the abstract model and the solution to the original model.

Different decisions about these two questions can lead to different tradeoffs between efficiency and solution quality. Conservative approaches (Dean, Givan, & Leach, 1997) aggregate states if they have similar reward and transition functions. This approach permits reasonable bounds on the relationship between the optimal solution to the aggregated model and the optimal solution to the original model. Unfortunately, such methods also fail to capture some of the intuitive notion of an aggregated state. For example, it would seem that all states within a particular room of a house should, at some level of abstraction, be grouped together as a single "room" state. The fact that actions might have different effects in different parts of the room would prevent this.

Bolder methods of aggregating states can be quite successful in practice, but are subject to some dangerous pitfalls. If states that are not similar in the right way are clustered together, it may be impossible to use the solution to the aggregated problem to reconstruct a useful policy for the original problem. Things get worse in a reinforcement learning context. Since the transition probabilities for all of the states within an aggregated state are not known *a priori*, the agent must use experiences from any of the states represented by the aggregated state to determine a value function. If the aggregated states are dissimilar, TD or Q-learning may not converge to a consistent value function, resulting in divergence or oscillation.

While state abstraction has many pitfalls, it ultimately must play a role in the toolbox of MDP solution methods. Since no two situations are ever truly the same, some implicit temporal abstraction is performed whenever a model is constructed and irrelevant features are discarded. Moreover, people appear to use some form of state abstraction in their own representations of the environment. Nevertheless, for the sake of clarity, state abstraction is treated as an orthogonal issue to the ideas presented in this dissertation and speculation on synergies between temporal abstraction and state space abstraction is reserved for the future work section.

2.6.2 Value Function Approximation

Value function approximation is a technique that has been applied to almost every direct MDP solution method. Value function approximation uses a parameterized function, $F(W, s)$ or $F(W, s, a)$, to represent a value function or a Q-function, where $W = (w_1 \ldots w_k)$ is a set of weights or parameters. Typical candidates for F could be a neural network or a multidimensional lattice with interpolation between lattice points. Off-line methods generally work in two phases, a dynamic programming phase and a function approximation phase. For example, value iteration with a neural network would start with some random set of weights, w^0, which imply a V^0. Some set of candidate states, G, would be selected and pushed through the Bellman equation to produce a set of values representative of V^1. A neural network would then be trained on this representative set of states to learn a new set of weights, w^1 that approximates V^1 and that, hopefully, generalizes well to cover the states not in G.

Value iteration with neural networks is not guaranteed to converge. However, Baird (1995) shows that by modifying the weight adjustment rule for the neural network, it is possible to ensure convergence to a local optimum. However, this modification tends to produce very slow convergence in practice and there do not appear to be any examples of the successful application of this method to large problems. Gordon (1995) shows that for a class of function approximation methods called *averagers*, value iteration based methods will converge to a stable solution. Averagers use a fixed candidate set, G, and compute the value of states outside of G as a convex combination of the values of the states in G. The point to which value function approximation will converge with an averager is well characterized. This appears to be a useful method for problems that are amenable to this form of value function approximation: problems with relatively smooth, locally approximately linear value functions.

Value function approximation for temporal difference learning or Q-learning adjusts the weight in the function approximator by taking a gradient. For TD(0), the update rule is

$$w^i \leftarrow w^{i-1} + \alpha\beta[F(s,W) + r - F(s',W)]\nabla_w F(s,W)$$

and for Q-learning it is

$$w^i \leftarrow w^{i+1} + \alpha\beta[F(s,a,W) + r - F(s',a,W)]\nabla_w F(s,a,W)$$

Unfortunately, value function approximation is subject to even more difficulties when it is used with reinforcement learning. Convergence is guaranteed only in a few special cases (Bertsekas & Tsitsiklis, 1996), and simple examples of divergence are available with common function approximators such as neural networks (Boyan & Moore, 1995). Nevertheless, function approximation has been used successfully in some high-profile applications (Tesauro, 1989; Zhang & Dietterich, 1995; Crites & Barto, 1996), and it is a promising area for investigation since it captures the intuitive idea that value functions ought to have compact representations. Value function approximation is a largely orthogonal issue for most of the topics covered in this dissertation, although some prospects for the use of function approximators are discussed for the temporal abstraction algorithms in the following two chapters. The application of value function approximation to the HAM and decomposition algorithms presented here is discussed as future work.

2.6.3 Temporal Abstraction

Temporal abstraction aims to manage large MDP state spaces by introducing more complicated actions that take place on larger time scales. Simple examples of abstract actions could be the action of doing something until a condition is satisfied, e.g., moving forward until some point is reached. This type of action can have a stochastic time duration as well a distribution over possible next states. The notion of temporal abstraction was introduced to the AI community in 1995 in the context of reinforcement learning by Sutton (1995) and in an off-line context by Dean and Lin (1995).

Chapter 3 shows that the AI notion of temporal abstraction corresponds to a less well-known, but well-documented (Puterman, 1994) generalization of MDPs called Semi-Markov Decision Problems (SMDPs). This relationship allows general and powerful theorems about SMDPs to be applied to temporal abstraction and provides new insight into AI algorithms that have used temporal abstraction.

While SMDPs generally have been used for event-driven systems where exogenous events drive the environment into a state that requires a control input, their use from an AI perspective permits the modeling of complex behaviors as single actions. Chapter 5 takes this approach to an extreme, where the HAM language for describing complex actions with subroutine calls is introduced. The most intriguing aspect of this language is that it permits *partial* or *abstract* descriptions of actions that can be optimally refined.

Temporal abstraction also plays an indirect role in the hierarchical decomposition algorithms of Chapter 6, where policies defined over entire regions of the state space are manipulated as if they were primitive actions. This form of temporal abstraction plays an important role in the hierarchical decomposition and solution of MDPs.

Although temporal abstraction does not reduce directly the size of the state space in the way state space abstraction does, it can reduce indirectly the state space through the introduction of behaviors that skip over large parts of the space. For example, if an agent uses temporally extended actions that repeat until certain conditions are met, the states in which no stopping conditions are satisfied are effectively removed from the state space since the agent is always executing some predefined behavior in those states. This type of state space reduction is discussed in the following chapters and exploited in the HAM algorithms of Chapter 5.

A final, appealing property of temporal abstraction is that it is less prone to the difficulties encountered by state space abstraction and value function approximation when it is used in a reinforcement learning context.

Chapter 3

Temporal Abstraction

The previous chapter presented the standard MDP framework, which implicitly assumes that all actions in all states have a uniform time duration. This chapter addresses the question of temporal abstraction in MDPs, in which actions can have complex effects that take place over a variable number of time steps. Numerous techniques that have been developed for this type of reasoning in MDPs are summarized and unified herein, including Sutton's mixture models. A major conceptual contribution of this chapter is the understanding of many methods for temporal abstraction as transformations from a policy over a region of an MDP to an action in an equivalent semi-Markov decision process (SMDP).[1] This chapter also contains a convergence proof for Q-learning in SMDPs.

3.1 Semi-Markov Decision Processes

The standard MDP framework models the environment at the granularity of a single time step. Time is not mentioned explicitly because all actions are presumed to take place at the same time scale. Consider the simple navigation problem in Figure 3.1. Variations of this problem will be a running example. Two rooms are connected by a "door," with a start state in one room and a reward state in the other. The problem of finding the optimal policy for reaching the reward state easily can be stated as a Markov decision problem.

[1]The connection between SMDPs and the author's previous work on temporal abstraction was first suggested by Sridhar Mahadevan at the AAAI 1996 Fall Symposium.

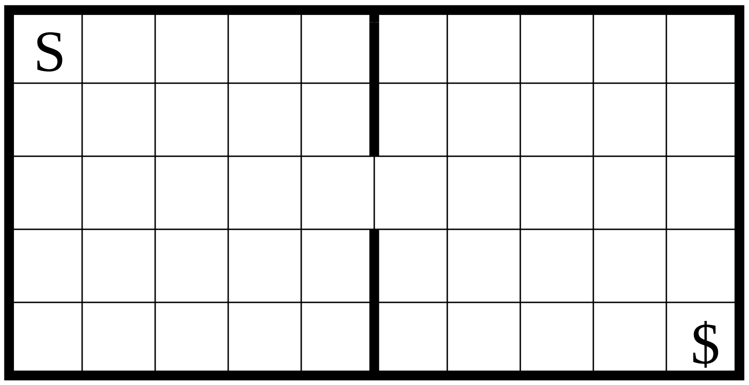

Figure 3.1: A simple navigation problem. Similar representations are used in Precup and Sutton (1997) and Hauskrecht et al. (1998).

One way in which the problem can become a bit more interesting is to suppose that the policy is to be carried out by a robot with unreliable actuators. When the robot attempts to move in the desired direction, it successfully activates its motors with probability p, but with probability $1-p$, it will be forced to reset its actuators and try again on the next time step. After each time step, the robot can try again, so the probability of an action taking n time steps is $(1-p)^{n-1}p$.

Examples of actions that have variable durations with this type of distribution are quite common in the real world — consider how long it takes to find the right key on one's keychain in the dark. Semi-Markov decision processes (SMDPs) are a generalization of Markov Decision processes that handle this and more general time distributions for actions. Specifically, an SMDP is just like an ordinary MDP, with the difference that transitions may have a stochastic time duration. Alternatively, the process can be thought of as passing through some number of *uncontrolled* states where the agent has no influence over the progression of the process before returning to a *controlled* state. As long as the process is Markovian in the controlled states, these two interpretations are equivalent.

The formal MDP definition is generalized for SMDPs as follows: An additional element $\mathcal{F}$ is added to the standard MDP components to define an SMDP as $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, where $\mathcal{F}$ is a function that defines the cumulative probability distribution over the number of time steps until the next controlled state. The probability that the next controlled state has been reached by time t when action a is taken in state s is written: $\mathcal{F}(t|s,a)$. In general, t may be real valued and range from 0 to $+\infty$. It is assumed that the distribution over the number of time steps taken does not change on subsequent trials of a in s. The focus of this chapter will be the case where the state space and action space are finite, so $\mathcal{F}$ can be thought of as a table indexed by pairs, $[s, a]$, where each table entry describes a probability distribution over t. Clearly, MDPs are a special case of SMDPs, where $\mathcal{F}$ is a step function with a discontinuous jump from 0 to 1 at $t = 1$.

The reward function must be reinterpreted to account for the fact that rewards may arrive at different times during the delay between controlled states. For example, on a transition from state s to s', rewards received immediately upon exit from state s should be treated differently from rewards received upon entry in state s'. It is assumed that rewards arrive stochastically and are weighted in some manner upon arrival. $\rho(s, a, t)$ describes the mean, weighted, accrued reward rate on transitions from s to s' under action a and taking time t, and

$$\mathcal{R}(s, a) = \int_0^\infty \int_0^t \rho(s, a, t) dt d\mathcal{F}(t|s, a)$$

is a convenient way of summarizing the expected reward associate with a particular state and action. It is a requirement of the SMDP framework that $\mathcal{R}(s, a)$ is finite. As with MDPs, it is possible to include the destination state, s', in the specification of $\mathcal{R}$, and ρ, but s' can be averaged out to produce the standard form. Nevertheless, it is sometimes useful to leave the destination state in the definition of $\mathcal{R}$ when it is necessary to disambiguate the reward received upon transition to the next controlled state and the reward received upon transition to the next possibly uncontrolled state. It also is possible to include a destination state in the time distribution, $\mathcal{F}(t|s, a, s')$. The destination state also can be removed in this case, but this operation requires some slight modifications to the transition model as well.

Although this formulation may seem somewhat dry and mathematical now, SMDPs capture the essence of temporal abstraction. Each action is not necessarily an atomic event, but may be the launching point for a complex series of events that take place over a variable number of time steps. The transition model, $\mathcal{T}$, and the time distribution, $\mathcal{F}$, give a potentially compact description of the effects of this type of action. The following section demonstrates that SMDPs are no harder to manipulate and solve than regular MDPs.

3.2 Algorithms for SMDPs

Optimality for SMDPs can be defined using the same optimality criteria as used for standard MDPs. For discounted total reward optimality, which is the focus here, the Bellman equation becomes:

$$V^*(s) = \max_{a \in A} \mathcal{R}(s,a) + \sum_{s' \in S} \mathcal{T}(s,a,s') \int_0^\infty \beta^t V^*(s') \mathcal{F}(t|s,a) dt$$

The maximizing actions of this equation define a class of optimal policies, Π^*. Since the state values are essentially constants whenever the integral is computed, $V^*(s')$ can be pulled out of the integral yielding:

$$V^*(s) = \max_{a \in A} \mathcal{R}(s,a) + \beta(s,a) \sum_{s' \in S} \mathcal{T}(s,a,s') V^*(s')$$

where

$$\beta(s,a) = \int_0^\infty \beta^t \mathcal{F}(t|s,a) dt, \tag{3.1}$$

and is interpreted as a discount rate that varies with the state and action. The dynamic programming operator for a particular action is then:

$$J_a(s) = \mathcal{R}(s,a) + \beta(s,a) \sum_{s' \in S} \mathcal{T}(s,a,s') V(s')$$

and in general:

$$J(s) = \max_{a} [\mathcal{R}(s,a) + \beta(s,a) \sum_{s' \in S} \mathcal{T}(s,a,s') V(s')]$$

It is easy to see that the dynamic programming operators described in Chapter 2 are contractions at rate $\beta^{\max} = \max_{s,a} \beta(s,a)$, and generalizations of the standard MDP algorithms (Chapter 2), such as linear programming, value iteration and policy iteration to SMDPs are well-known (White, 1993; Puterman, 1994).

On-line algorithms, including Q-learning and Temporal Difference learning have also been adapted to SMDPs (Bradtke & Duff, 1995), although formal convergence results do not appear to have been published. Q-learning is modified for SMDPs as follows: On a transition from state s to s' under action a that has taken time t and received reward r (which is assumed to be the appropriately weighted sum of rewards received during t):

$$Q(s,a)^i \leftarrow Q^{i-1}(s,a) + \alpha^i(s,a)(r + \beta^t V^{i-1}(s') - Q^{i-1}(s,a))$$

where, as in Chapter 2, $\alpha^i(s,a)$ is a learning rate that can depend upon the state, action, and number of learning steps taken so far.

When $|\mathcal{A}_s| = 1$ for all s, i.e. there are no controlled states or the policy is fixed, this results in the $TD(0)$ learning rule for SMDPs. The convergence of this modified form of Q-learning for SMDPs is proved here, based directly on the convergence proof of Q-learning in Jaakkola et al. (1994).

Theorem 1 *(Jaakkola et al., 1994) A random iterative process* $\Delta_{i+1}(x) = [1-\alpha_i(x)]\Delta_i(x) + \alpha_i(x)G_i(x)$ *converges to zero with probability one (w.p.1) under the following assumptions:*

1. $x \in S$, *where* S *is a finite set.*

2. $\sum_i \alpha_i = \infty, \sum_i \alpha_i^2(x) < \infty$ *uniformly over* s *and* a *w.p.1..*

3. $Var\{G_i(x)|P_i, \alpha_i\} \leq C(1 + \|\Delta_i\|_W)^2$, *where* C *is some constant.*

4. $\|E\{G_i(x)|P_i, \alpha_i\}\|_W \leq \beta\|\Delta_i\|_w$, *where* $\gamma \in (0,1)$.

Here $P_i = \{x_i, x_{i-1}, \ldots, F_{i-1}, \ldots, \alpha_{i-1}, \ldots\}$ *i.e. the complete history up to step* i. $G_i(x)$ *and* $\alpha_i(x)$, *are allowed to depend on the past, and* $\alpha_i(x)$ *is assumed to be nonnegative. The notation* $\|\cdot\|_W$ *refers to some weighted maximum norm.*

This general stochastic process theorem uses the fact that G is a contraction *in expectation* and permits the following theorem on Q-learning for SMDPs.

Theorem 2 *The SMDP Q-learning rule converges to the optimal* $Q^*(s,a)$ *values if*

1. *The state and action spaces are finite.*

2. $\sum_i \alpha_i(s,a) = \infty$ *and* $\sum_i \alpha_i^2(s,a) < \infty$ *uniformly over* s *and* a *w.p.1.*

3. $Var\{r\}$ *is finite.*

4. $0 < \beta^{max} < 1$

Proof: The proof follows closely the convergence proof for Q-learning in Jaakkola et al. (1994). The SMDP Q-learning update rule is shown to be a process of the appropriate form where $\Delta_t(s,a) = Q_t(s,a) - Q^*(s,a)$ and

$$G_i(s,a) = r + \beta^t + V(s') - Q^*(s,a).$$

This mirrors the Q-learning update rule with a constant equaling the optimal $Q^*(s,a)$ subtracted off. Since Δ_i is the difference between the estimated and optimal Q-values, convergence of Δ_i to zero will prove the convergence of Q-learning for SMDPs.

Condition 1 and 2 of this theorem satisfy conditions 1 of 2, respectively, of Theorem 1. Condition 3 of this theorem follows from the fact that $\mathcal{R}(s,a)$ is bounded and satisfies condition 3 of Theorem 1.

What remains to be shown is that G is a contraction in expectation, which is done as follows:

$$\begin{aligned} E\{G_t(s,a)\} &= \mathcal{R}(s,a) + \left[\sum_{s'} \mathcal{T}(s,a,s') \int_0^\infty \beta^t V^i(s') d\mathcal{F}(t|s,a)\right] - Q^*(s,a) \\ &= J_a V^i - Q^*(s,a) \end{aligned}$$

where J is the value iteration operator for action a as in Chapter 2. To satisfy property 3 of Theorem 1, it must be the case that:

$$\|J_a V^i - Q^*(s,a)\|_W \quad \leq \quad \beta^{\max} \|Q^i(s,a) - Q^*(s,a)\|_W$$

which is true since the J_a operator applied to V^i is a contraction with rate $\beta^{\max}$ and fixed point at $Q^*(s,a)$. ■

As with the proof for regular Q-learning, this proof requires that the learning rate does not decay too quickly, and requires a finite state and action space, which is a restriction on the standard MDP definition. The learning rate requirements implicitly require that every action be tried infinitely often in every state. The SMDP version also requires $\beta^{\max} < 1$, while the MDP version can tolerate some cases where $\beta = 1$. The SMDP result can be generalized if zero-discount cycles are forbidden.

3.3 Equivalence of SMDPs

In the previous sections, the general representation of an SMDP with an (almost) arbitrary distribution over time durations between transitions was reduced to a representation where each transition in the model took a single step, but where different state-action combinations could have different discount factors (3.1). When it is possible to integrate over the reward and duration functions to produce the equivalent discrete time representation, the representation of and computations required by SMDPs are greatly simplified.

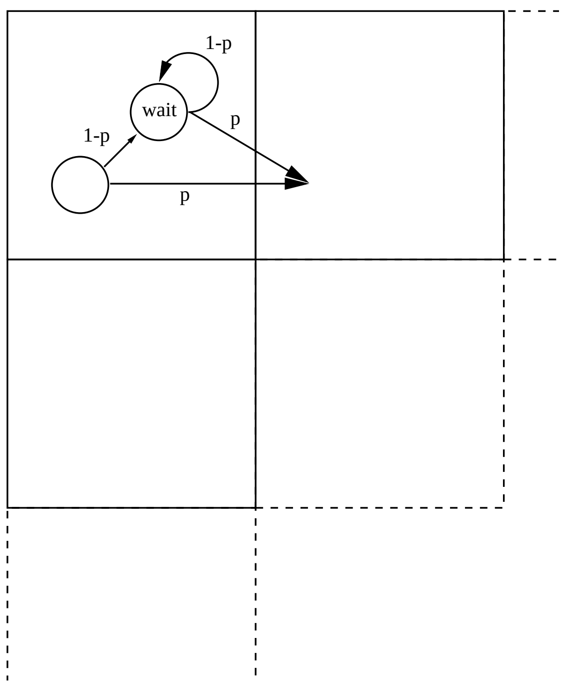

Figure 3.2: An uncontrolled wait states is inserted into the simple top-left corner of the navigation problem.

This section further explores the notion of equivalence between different representations of SMDPs with the aim of developing a definition of equivalence that can be applied to SMDPs with different numbers of uncontrolled states. Suppose, for example, there are two models of the robot's behavior, one that uses different discount factors to reflect the unreliability of the robot's motor, and one that uses an uncontrolled "wait" state between transitions as in Figure 3.2. These two models will be equivalent if they can be used interchangeably for the purposes of planning, i.e. if there is mapping between supersets of the controlled states such that for any state-action pair, the reward and transition functions from one model can be subsituted into the second without changing the Bellman equation.

Consider two SMDPs, $M^1 = (\mathcal{S}^1, \mathcal{A}^1, \mathcal{T}^1, \mathcal{R}^1, \beta^1)$ and $M^2 = (\mathcal{S}^2, \mathcal{A}^2, \mathcal{T}^2, \mathcal{R}^2, \beta^2)$, where the time distribution, $\mathcal{F}$, has been replaced with the equivalent state-action dependent discount functions. Let $\mathcal{S}^1_C \subseteq \mathcal{S}^1$ and $S^2_C \subseteq S^2$ be the sets of controlled states in S^1 and S^2, respectively. M^1 and M^2 are *equivalent* if there exists a function, f^{12}, that maps states of M^1 to states of M^2, and actions of A^1 to actions of A^2, a function, f^{21}, that maps states of M^2 to states of M^1, and actions of A^2 to actions of A^1 and sets $\mathcal{S}^1_C \subseteq \mathcal{S}^1_E \subseteq \mathcal{S}^1$ and $\mathcal{S}^2_C \subseteq \mathcal{S}^2_E \subseteq \mathcal{S}^2$, such that $\forall s \in S^1_E$ and $\forall a \in \mathcal{A}^1_s$:

$$\begin{aligned}&\beta^1(s,a)\sum_{s'\in\mathcal{S}^1_E}\mathcal{T}^1(s,a,s')[\mathcal{R}^1(s,a,s')+V(s')]\\ &\quad=\beta^2(f^{12}(s),a)\sum_{s'\in\mathcal{S}^2_E}\mathcal{T}^2(f^{12}(s),a,s')[\mathcal{R}^2(f^{12}(s),a,s')+V(f^{21}(s'))]\end{aligned}$$

and $\forall s \in S^2_E$ and $\forall a \in \mathcal{A}^2_s$:

$$\begin{aligned}&\beta^2(s,a)\sum_{s'\in\mathcal{S}^2_E}\mathcal{T}^2(s,a,s')[\mathcal{R}^2(s,a,s')+V(s')]\\ &\quad=\beta^1(f^{21}(s),a)\sum_{s'\in\mathcal{S}^1_E}\mathcal{T}^1(f^{21}(s),a,s')[\mathcal{R}^1(f^{21}(s),a,s')+V(f^{12}(s'))]\end{aligned}$$

(Note that f^{12} and f^{21} are overloaded in the sense that they are defined on both actions and states.) This definition of equivalence would, of course, need to be changed for different optimality criteria.

Consider again the navigation problem of Figure 3.1. Call the top, left state, s_1, and the state one square to the right, s_2. If a robot in s_1 has an unreliable actuator and is trying to move right, this can be modeled with a wait state as in Figure 3.2.[2] Call the wait state, s_{1w}, and call the action that moves the robot to the right, a_1. The value of moving right is:

$$Q(s_1,a_1)=\mathcal{R}(s_1,a_1)+\beta(pV(s_2)+(1-p)V(s_{1w}))$$

and

$$V(s_{1w})=\beta(pV(s_2)+(1-p)V(s_{1w})).$$

The equivalent model with no wait state can be determined by solving the second equation for $V(s_{1w})$ and substituting into the first equation, yielding

$$Q(s_1,a_1)=\mathcal{R}(s_1,a_1)+\frac{\beta p-\beta^2p-\beta^2p^2+\beta p-\beta p^2}{1-\beta-\beta p}V(s_2),$$

which can be thought of as a model with a probability 1 transition to s_2 and a discount factor of $\frac{\beta p-\beta^2p-\beta^2p^2+\beta p-\beta p^2}{1-\beta-\beta p}$. This shows that the wait-state representation of the robot's

[2]Note that this presentation is somewhat simplified for expository purposes. It assumes that there is no uncertainty in the next state and considers only a single action that moves the robot to the right. If the robot could move in each of the four coordinate directions, then there would need to be additional wait states for each direction.

actuator uncertainty is equivalent to a representation that uses a different discount factor. This somewhat awkward definition of equivalence comes in handy here because it permits an equivalence statement about models with different numbers of states by requiring interchangeability of the right-hand-side of the Bellman equation for only a superset of the controlled states and not the entire state space.

Equivalent SMDPs have several straightforward properties which are stated here as theorems to underscore their importance:

Theorem 3 *Equivalence of SMDPs is a symmetric relation.*

Proof: This follows immediately from the definition of equivalence. ■

In the following theorem, which establishes the relationship between policies of equivalent SMDPs, the functions describing the mappings between two SMDPs will be overloaded further to map from policies to policies. $f^{12}(\pi)$ maps policies defined over states in $\mathcal{M}^1$ to a policies defined over states of $\mathcal{M}^2$ such that if $\pi^2 = f^{12}(\pi^1)$, then $\pi^2(s) = f^{12}(\pi^1(f^{21}(s)))$.

Theorem 4 *For any equivalent SMDPs, $\mathcal{M}^1$ and $\mathcal{M}^2$, where f^{12} is the mapping from elements of $\mathcal{M}^1$ to elements of $\mathcal{M}^2$ that satisfies the definition of equivalence, $V_{\pi^1}(s) = V_{f^{12}(\pi^1)}(f^{12}(s))$, for all π^1 defined on $\mathcal{M}^1$ and all s in $\mathcal{S}^1_E$.*

Proof: V_{π^1} can be determined by solving the system of equations that results when the actions of π^1 are assigned to the states of $\mathcal{S}^1$. By the definition of equivalence, $V_{f^{12}(\pi^1)}$ can be determined by solving a system of equations that is isomorphic to the system for V_{π^1} and where f^{12} describes the renaming of the variables from one system to the other. Thus, the solutions to the systems must be identical up to a renaming of the variables. ■

This theorem verifies the intuition that a policy for one MDP has a corresponding policy in all equivalent MDPs. This also applies to optimal policies and the resulting optimal value functions:

Corollary 1 *For any equivalent SMDPs, $\mathcal{M}^1$ and $\mathcal{M}^2$, $V^*(s) = V^*(f^{12}(s))$, $\forall s \in \mathcal{S}^1_E$.*

Corollary 2 *For any equivalent SMDPs, $\mathcal{M}^1$ and $\mathcal{M}^2$, if $\pi^* \in \Pi^{1*}$ then $f_{12}(\pi^*) \in \Pi^{2*}$.*

It is possible to define a *variable discount MDP* (VDMDP) as a variation on the standard MDP definition, but with a discount factor, $\beta(s, a)$, that can depend upon the state and action. Note that the optimality criterion of discounted total reward is, in some sense, implicit in the this definition. The previous section showed that any SMDP can be expressed as a VDMDP. In fact, the two classes of problems are equivalent:

Theorem 5 *SMDPs and VDMDPs are equivalent under the discounted total reward optimality criterion.*

Proof: The subset relationship is established in both directions:

1. SMDPs $\subseteq$ VDMDPs: This was established as part of the definition of SMDPs (3.1).
2. VDMDPs $\subseteq$ SMDPs: Any VDMP can be transformed into an SMDP, by setting β for the SMDP as $\beta^{\max} = \max_{s,a} \beta(s,a)$ and setting $\mathcal{F}(t|s,a)$ to be a step function that jumps from 0 to 1 at $t = \frac{\log \beta(s,a)}{\log \beta^{\max}}$ since:

$$\mathcal{R}(s,a) + \beta(s,a) \sum_{s'} \mathcal{T}(s,a,s')V(s') = \mathcal{R}(s,a) + \beta^{\frac{\log \beta(s,a)}{\log \beta^{max}}} \sum_{s'} T(s,a,s')V(s'). \blacksquare$$

More generally,

Theorem 6 *For any Markov decision process for which the Bellman equation for each state-action pair can be expressed as*

$$Q(s,a) = \sum_{s'} c_{s,a,s'} V(s') + k_{(s,a)}$$

where, $0 < \sum_{s'} c_{s,a,s'} < 1$ *and* $c_{s,a,s'} \geq 0$ *for all* s, a *and* s', *there exists an equivalent SMDP.*

Proof: Construct the new MDP reward function by assigning $\mathcal{R}(s,a)$ to the constant of the linear function for state s and action a. The discount function for the SMDP is $\beta(s,a) = \sum_{s'} c_{s,a,s'}$ and transition function is just $\mathcal{T}(s,a,s') = \frac{c_{s,a,s'}}{\beta(s,a)}$. $\blacksquare$

To summarize, the preceding two theorems show that there are three equivalent representations for SMDP actions: a transition function with a fixed discount factor and a time distribution, a transition function with a variable discount factor and a fixed time step size; a linear value relationship between states such that the coefficients sum to be less than 1. These results summarize and generalize observations made in Puterman (1994), Bradtke and Duff (1995), and White (1993).

The following results formally establish the intuition that any uncontrolled states can be removed from any SMDP to produce an equivalent SMDP:

Lemma 1 *For any SMDP, $\mathcal{M}^1 = (\mathcal{S}^1, \mathcal{A}^1, \mathcal{T}^1, \mathcal{R}^1, \beta^1)$ with $n > 0$ uncontrolled states, at least one of which is non-absorbing, there exists an equivalent SMDP with $n-1$ uncontrolled states.*

Proof: Create a new SMDP, $\mathcal{M}^2 = (\mathcal{S}^2, \mathcal{A}^2, \mathcal{T}^2, \mathcal{R}^2, \beta^2)$, by modifying the original, so that $\mathcal{S}^2 = \mathcal{S}^1 - s_i$, where s_i is some non-absorbing uncontrolled state. The new transition function is defined as follows:

$$\mathcal{T}^2(s,a,s') = \mathcal{T}^1(s,a,s') + \frac{\mathcal{T}^1(s,a,s_i)\mathcal{T}^1(s_i,s')}{1 - \quad T^1(s_i,s_i)}.$$

The reward function is defined as follows [3]:

$$\mathcal{R}^2(s,a,s') = \frac{\mathcal{T}^1(s,a,s')\mathcal{R}^1(s,a,s') + \frac{\mathcal{T}^1(s,a,s_i)\mathcal{T}^1(s_i,s')\mathcal{T}^1(s_i,s_i)\mathcal{R}^1(s_i,s_i)}{[1-\mathcal{T}^1(s_1,s_i)][1-\mathcal{T}^1(s_i,s_i)\beta^1(s_i,s_i)]} + \frac{\mathcal{T}^1(s,a,s_i)\mathcal{T}^1(s_i,s')\mathcal{R}^1(s_i,s')}{1-\mathcal{T}^1(s_i,s_i)\beta^1(s_i,s_i)}}{\mathcal{T}^2(s,a,s')},$$

and the discount factor is defined as follows[4]:

$$\beta^2(s,a,s') = \frac{\mathcal{T}^1(s,a,s')\beta^1(s,a,s') + \frac{\mathcal{T}^1(s,a,s_i)\mathcal{T}^1(s_i,s')\beta^1(s_i,s')}{1-\mathcal{T}^1(s_i,s_i)\beta^1(s_i,s_i)}}{\mathcal{T}^2(s,a,s')}.$$

The Bellman equation for $\mathcal{M}^2$ is equivalent (after some algebra) to the Bellman equation for M^1 when the Bellman equation for s_i,

$$V(s_i) = \frac{\mathcal{T}^1(s_i,s_i)\mathcal{R}^1(s_i,s_i) + \sum_{s' \in \{\mathcal{S}^1 - s_i\}} \mathcal{T}^1(s_i,s')[\mathcal{R}^1(s_i,s') + \beta^1(s_i,s')V(s')]}{1 - \mathcal{T}^1(s_i,s_i)\beta^1(s_i,s_i)},$$

is substituted for $V(s_i)$ in the Bellman equation for any s_j, $j \neq i$, in $\mathcal{S}^1$. ■

An *absorbing cycle* is a cycle from which the probability of exit is 0. Note that if there are no absorbing cycles in a model, the removal of a state by the above lemma cannot create one.

Theorem 7 *For any SMDP with n uncontrolled states and no absorbing cycles, there exists an equivalent SMDP with 0 uncontrolled states.*

Proof: The proof follows trivially by induction on n using the previous lemma.■

There is a strong conceptual connection between this notion of equivalence and the notion of stochastic bisimulation homogeneity (Dean & Givan, 1997). The main difference is that the Dean and Givan work focused on aggregating similar states, while the focus here is on the complete removal of irrelevant states.

[3]The $\mathcal{T}^2$ in the denominator is *not* a typographical error.
[4]The $\mathcal{T}^2$ in the denominator is correct here too.

3.4 Temporally extended actions

The preceding sections described SMDPs and the basic machinery used to remove uncontrolled states and otherwise transform between different, but equivalent, representations of SMDP transition functions. This section discusses several approaches to temporal abstraction that have been proposed in the literature and shows that these superficially different approaches have a fundamental connection: the transformation of a policy defined over a region of the state space into an action in an SMDP.

The notion of a temporally extended or abstract action has appeared in many places in the MDP literature. The earliest may be in Forestier and Varaiya (1978), where the approach was more from the perspective of control theory, with a focus on the average reward optimality criterion. Nevertheless, Forestier and Varaiya uncovered basic insights that were subsequently rediscovered by many authors, including this one (Parr, 1996).

Subsequent research in the artificial intelligence community has focused mainly on reinforcement learning applications, but the basic ideas are quite similar. While none of the authors used the SMDP framework, they had a common vision of temporal abstraction that can be related directly to SMDPs. These relationships are detailed below, along with some commentary on the advantages afforded by temporal abstraction in each case.

3.4.1 Origins in Control Theory

Control theorists tend to think of MDPs modeling industrial plants instead of agents wandering through an environment. Forestier and Varaiya addressed the issue of controlling a plant for which the state space is presumed to be very large, large enough that finding the optimal action for every state would be impractical. Instead of solving the full MDP directly, a simplification of the problem is proposed. A set of *candidate policies* and a set of *boundary conditions* are identified. The boundary conditions can be thought of as dangerous states that must be avoided, and the candidate policies can be thought of as heuristics designed to keep the system out of the dangerous states. Once a candidate policy is chosen, this policy will be active until the system reaches a boundary condition. When a boundary condition is reached, a new policy is selected from one of the candidate policies. Forestier and Varaiya showed that the problem of making the optimal assignment of policies to boundary states can be thought of as another MDP where the states correspond to the boundary conditions and the actions correspond to the candidate policies.

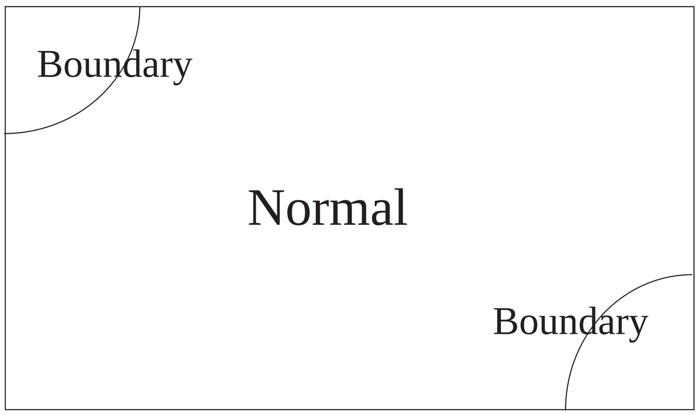

Figure 3.3: A state space with boundary regions.

This approach is not that far from what could happen if one were to design a controller for a real system that was previously controlled by humans. People employ rules of thumb, which would correspond to candidate policies. They also are trained to identify boundary conditions which would require intervention of some kind. While it may not be possible to make the optimal decision for every possible state of the plant, it could be possible to switch optimally between the different rules of thumb used by humans in all of the circumstances that previously required human attention. Such a policy would be guaranteed to do at least as well at controlling the plant as the people who contributed the rules of thumb — and might do better.

Figure 3.3 shows the partitioning of a state space into normal and boundary conditions. Suppose that there are just two candidate policies, π_1 and π_2; then the problem becomes one where there are two controlled regions and two uncontrolled regions, as seen in Figure 3.4. The controlled regions correspond to the boundary areas in the original state space, and the uncontrolled regions correspond to the "normal" region of the original state space. When a candidate policy is selected in a boundary region and the system enters the normal region, it runs without intervention until another boundary state is reached.

In the language of SMDPs, each state in the boundary region becomes a controlled state. For each normal state, two uncontrolled states are created: one where the action recommended by π^1 is taken, and one where the action recommended by π^2 is taken. Theorem 7 can be applied to this process, and all of the uncontrolled states can be removed, yielding an SMDP where the number of states is equal to the number of states in just

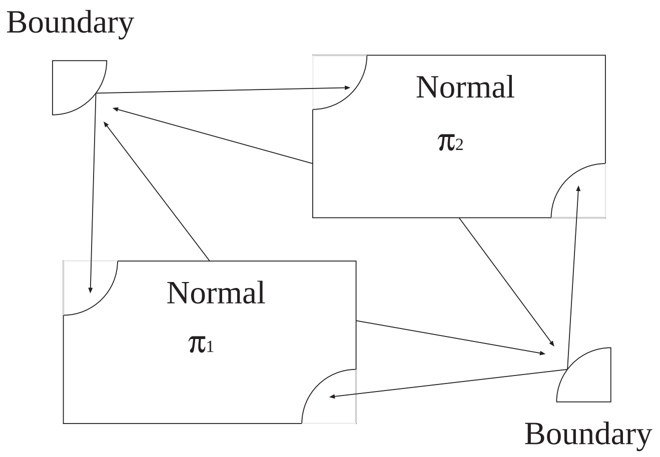

Figure 3.4: Process that switches between candidate policies.

the border regions. An action in this new SMDP corresponds to the decision to apply a candidate policy until the next boundary condition is reached. This transformation is significant because the solution to this smaller SMDP can be found with less computation than that required to solve the full MDP. Of course, there is a computational cost to be paid for the conversion. Algorithms for performing this type of conversion are described in the following chapter.

Another interesting property of this transformation is that while the optimal SMDP policy must be stationary, the implementation of this policy may be non-stationary and suboptimal at the level of the original MDP. Suppose that $\pi^1(s_1) \neq \pi^2(s_1)$ for some s_1 in the "normal" region, then the action taken in state s_1 will depend upon which policy was adopted at the previous boundary state. This is a natural consequence of abstraction when the language of the more abstract problem (the SMDP) is not the same as the language of the low-level problem (the original MDP). Fortunately, the relationship between the problems is still very tight. The solution to the SMDP corresponds to an actual, sensible policy in the original problem, a fact that is not always guaranteed when other methods, such as state abstraction, are used. Moreover, the value relationship between states in the original problem and states in the SMDP is clear: the value assigned to an abstract action in the SMDP is the true expected value of following the corresponding policy in the original problem. This is in contrast to value function approximation or state aggregation methods, which generally provide only loose guarantees on the value relationship between the solution to the abstract problem and the solution to the original problem.

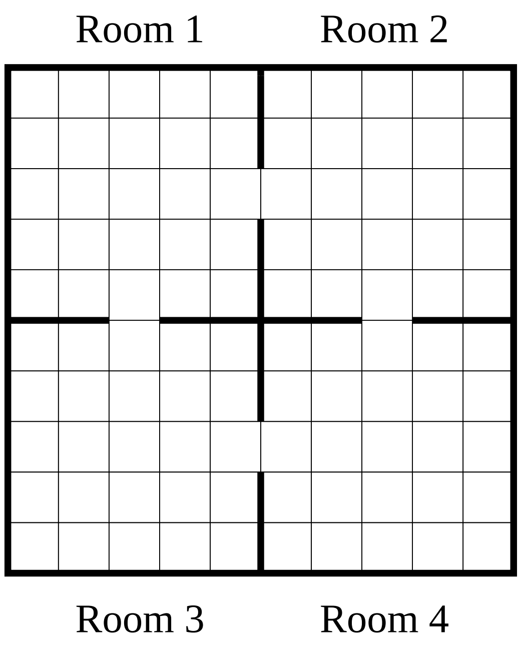

Figure 3.5: A four-room version of the navigation problem.

3.4.2 Freezing and Compiling Policies

In the AI community Dean and S.-H. Lin [5] introduced a method of policy compilation (Dean & Lin, 1995). This method, which was part of a larger approach to hierarchical solution methods for MDPs (see also Chapter 6), used the trick of "freezing" the policy for a particular region of the state space, then "compiling out" this region. It is then possible to determine the optimal policy for the remaining states, conditioned on the fact the policy in the removed region cannot change. This is shown for a four-room example in Figure 3.5 and Figure 3.6. The actions that connected the other regions to the removed region are replaced with "abstract actions," which are simply temporally extended actions that correspond to the execution of the frozen policy over the removed region. In the SMDP framework, when the policy in room 2 is fixed, the states in room 2 become uncontrolled and can be removed, according to Theorem 7, to produce a equivalent SMDP. Thus, the freezing and compiling process can be interpreted as a transformation to an SMDP in which the behavior in the removed region is expressed as an SMDP action.

[5] Two people named Lin recently made contributions to this area. L.-J. Lin worked in reinforcement learning with Tom Mitchell, while S.-H. Lin worked on the off-line methods of the previous subsection with Thomas Dean.

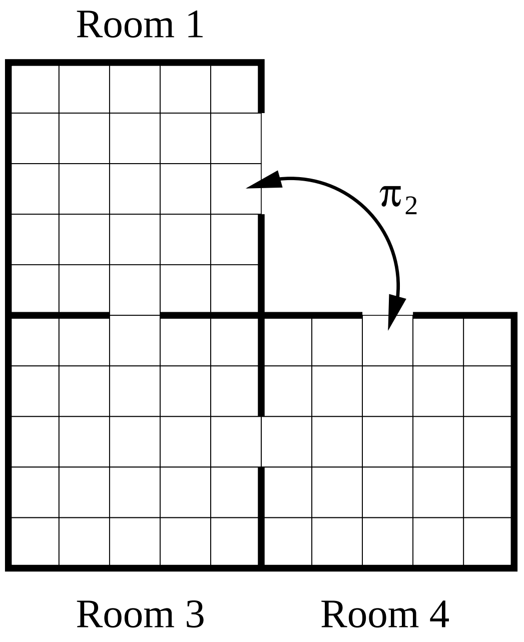

Figure 3.6: Policy π_2 is fixed for room 2 and then states of room 2 are removed, summarizing the effects of π_2 with SMDP actions defined at the connecting states in rooms 1 and 4.

This approach has much in common with the Forestier and Varaiya approach. The states at the borders between the rooms can be thought of boundary states and the interior of the rooms can be thought of as normal states. An important difference between the Dean and Lin approach and the Forestier and Varaiya approach is that the abstract actions were not simply determined *a priori* and used unchanged throughout the solution process in the former. The approach advocated by Dean and Lin works by first defining a partitioning of the state space into disjoint regions, $\mathcal{S} = G_1 \cup G_2 \cup \ldots \cup G_m$, defining a set of policies on these regions, solving the SMDP that combines theses policies, *updating* the policy for each region based upon the solution to the SMDP, and repeating until none of the individual region policies can be improved.

The mechanism for updating a policy based upon the SMDP solution deserves some further explanation and is best illustrated with the example in Figure 3.7. If each room in this model is a region, then the SMDP that combines the policies in each region contains just 8 states, the states connecting the rooms. The solution to this SMDP assigns values to each of these states. The policy in room 2, for example, can be updated in light of these value assignments, by solving a small MDP that contains just the states in room 2 and the states in rooms 1 and 4 that connect to room 2. The connecting states are treated

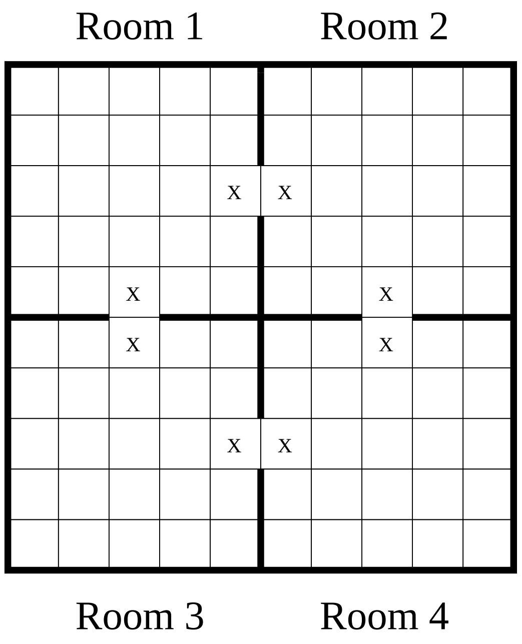

Figure 3.7: The four-room navigation MDP with Xs marking the states in the corresponding SMDP

as absorbing states, with values $V(s_1)$ and $V(s_4)$ determined by the SMDP solution as in Figure 3.8 and Figure 3.9.

Note that if the solution to the SMDP happens to assign the values of V^* in the original MDP to each of the corresponding states in the SMDP, then the policies that are determined for each room will be consistent with some optimal π^* for the original MDP. The algorithm can be stated in pseudocode as follows:

```
$\Pi$ = initial set of policies $\pi_1 \ldots \pi_m$ for $G_1 \ldots G_m$
$\hat{\pi} \leftarrow$ solution to the reduced SMDP induced by $\Pi$
Repeat until $\hat{\pi}$ cannot be improved
  For each $G_i$
    Determine the optimal policy for $G_i$ given $V_{\hat{\pi}}$
    Replace $\pi_i$ in $\Pi$ with the new policy
  $\hat{\pi} \leftarrow$ solution to SMDP induced by $\Pi$
```

This algorithm will provably converge to the optimal policy. The proof of this was originally established through a connection to the Dantzig-Wolfe decomposition of linear programs (Dantzig & Wolfe, 1960). However, in the SMDP interpretation, convergence to the optimal policy follows directly as a consequence of the convergence of policy iteration for SMDPs. To see this, consider again the navigation MDP and corresponding 8-state SMDP.

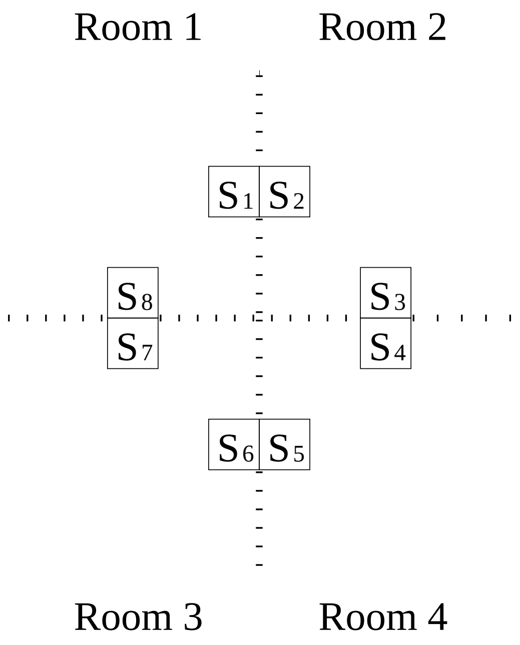

Figure 3.8: The new SMDP state space with states labeled $s_1 \ldots s_8$.

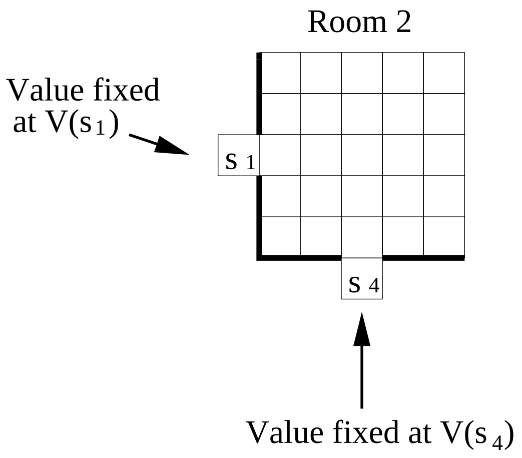

Figure 3.9: The policy for room 2 is revised based upon the solution to the SMDP.

The space of possible actions for each state in the SMDP is the space of possible policies in the enclosing room. The above algorithm uses the SMDP value function to construct a greedy policy for the SMDP, and repeats this process until the policy is optimal. This is precisely the policy iteration algorithm.

The understanding of this form of temporal abstraction as an operation on an SMDP also suggests other variants on the basic algorithm. For example, several policies could be defined for each region, as in the previous subsection, or the regions could be updated asynchronously. These options are explored further in Chapter 6.

3.4.3 Subtasks for neural networks

L.-J. Lin (1993) rediscovered the basic insight in Forestier and Varaiya (1978) in his work on reinforcement learning with neural networks for robot control. Lin took the approach of dividing complex tasks into subtasks and using Q-learning with neural networks as function approximators to solve separate learning problems, one for each subtask. A higher-level learning task was then solved in which the policies learned for the subtasks were treated as actions.

A subtask in Lin's framework was similar to the method used by Dean and S.-H. Lin for updating policies in regions. For each subtask, some states were identified as constituting achievement of the subtask and some guess was made about the value of those states. These states were then treated as absorbing states with values fixed at the guessed value, and an MDP was solved based upon this assumption. The policies resulting from these subtask solutions were then treated as actions.

The conversion from Lin's approach to the SMDP framework is not as clear as the Forestier and Varaiya approach simply because Lin was not as clear on when the agent could switch between policies. Lin recognized the need for some mechanism that would reduce the number of times switches could take place as an important element for simplifying the problem. Without such a mechanism, the agent might thrash during the learning process, switching between policies at every state. If the number of policies considered at each state is the same or greater than the number of actions, then the learning task would not be any easier.

Conceptually, Lin's approach can be seen as an attempt to transform the original learning task into an SMDP learning task, where learned policies become temporally abstract actions that can take place over several time steps. A mechanism that prevents the agent from thrashing can be seen as a mechanism that pushes the agent into an uncontrolled region of the SMDP space, much like the normal/boundary mechanism in Forestier and Varaiya (1978). The advantage of Lin's approach from a reinforcement learning standpoint was that several subtasks were often easier to learn, i.e. required fewer environment experiences and placed lighter demands on a function approximator, than a single monolithic task. If the subtasks were chosen well, combinations of the subtasks could quickly move the robot to useful parts of the state space that would otherwise require long periods of trial and error to reach in the original, low-level problem.

3.4.4 Mixture models

Mixture models (Sutton, 1995) take a somewhat more formal approach to L.-J. Lin's notion of subtasks. Sutton began with the perspective of a fixed policy, with the aim of learning a value function or predictive model. Sutton used the notion of a discounted transition matrix, which is equivalent to an SMDP transition matrix with the discount rate multiplied into the matrix. Together with a (discounted) reward function, these define a dynamic programming operator, J_π. Sutton observed that a family of different J operators could lead to the same V_π^∞. Since $V_\pi = J^\infty V_\pi$, then $J^2(V) = J(J(V))$, which is interpreted as a "2-step" operator, is an obvious candidate, but it turns out that there are many other possible J operators that work as well. Two dynamic programming operators, J_1 and J_2, are *interchangeable*[6] if:

$$J_1^\infty(V) = J_2^\infty(V)$$

for all V.

Clearly, interchangeability is reflexive, symmetric and transitive. Sutton made several useful observations about interchangeable backup operators:

1. If J_1 and J_2 are interchangeable, then $J_3(V) = J_1(J_2(V))$ is interchangeable with J_1 and J_2.

[6]Sutton uses the term *valid*, which is a unary predicate referring to a single dynamic programming operator. Implicit in his usage is the assumption that there is a fundamental model to which all others are compared. In controlled problems, where there is more than one possible action, it doesn't make sense to distinguish one as valid and others invalid in this way. The binary predicate *interchangeable* is used instead, which reflects that the two models can be used interchangeably without affecting the value function.

2. The weighted average of a set of operators, $J_1 \dots J_i$, produces an operator that is interchangeable with any of $J_1 \dots J_i$.

3. The above two observations may be applied to the pointwise dynamic programming operator, $J(s)$, for individual s, i.e. applied asynchronously.

4. For any operator, J_1, it is possible to create an interchangeable operator, J_β, that is a weighted combination of the i-step versions of J as i ranges from 1 to ∞ and where the weights can depend upon the trajectory taken through the model (full β-models, as described below).

The term β-model should not be confused with the SMDP convention of allowing the discount factor to vary at different states. Sutton did not use the SMDP framework, but used a similar approach in which the discount factor was multiplied directly into the transition and reward functions. This has the advantage of providing a cleaner notation in some cases, but has the disadvantage that the transition model is no longer a probability distribution since the discounted probabilities no longer sum to 1.

It turns out that each of Sutton's observations has a fairly straightforward interpretation in the SMDP framework using the theorems described in this chapter. This provides a framework that unifies Sutton's work with other temporal abstraction methods and provides a straightforward interpretation of Sutton's full β-model.

Multi-step operations

A dynamic programming operator propagates information to the current state from states one step in the future. Two successive applications of a dynamic programming operator can be interpreted as propagating information from two states in the future. In general, n applications a dynamic programming operator, J_π, propagate the effects of following policy, π, for n steps. The interchangeability of a mulit-step operators, $J_\pi^n V = J_\pi(J_\pi(\dots J_\pi V))$, with its constituent single step operator, J_π follows immediately from the contraction property of J_π, i.e., if J_π is a contraction, then J_π^n is necessarily a contraction to the same fixed point. Sutton shows that a new discounted transition matrix implementing this operator can be determined by a sequence of matrix multiplications.

The act of following a policy for n steps also can be interpreted in the SMDP framework. It can be seen as an *unrolling* of the model (think of loop unrolling by a

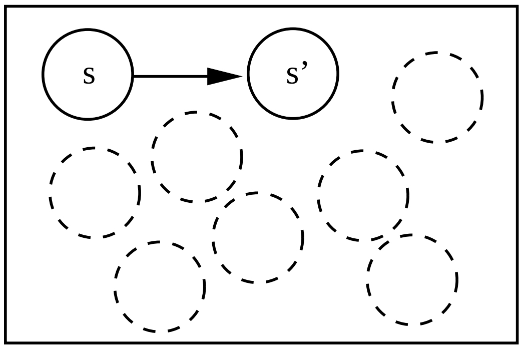

Figure 3.10: A model with a transition from s to s' highlighted.

compiler) so that n copies of the model are created. Wherever there was previously a transition from s to s', there is now a transition from $s_{t=i}$ to $s'_{t=i+1}$, except for $s_{t=n}$, which would have a transition to $s'_{t=1}$. See Figure 3.10 and Figure 3.11.

Using Lemma 1, all states for $t > 1$ can then be removed from the model to create an SMDP, the transition and reward function of which will express the n-step discounted probabilities and rewards that result when π is followed for n steps.

Composition of operations

That the composition of two interchangeable operators produces an equivalent operator also follows from the contraction property of the corresponding DP operators. Clearly, if J_1 and J_2 are both contractions with the same fixed point, then $J_3(V) = J_2(J_1(V))$ must also be a contraction to that fixed point.

There is also, however, an illuminating interpretation of this operation in the SMDP framework. The approach used is similar to that of unrolling, with the difference being that one copy of the states obeying J_1 is rolled into another copy of the states which obey J_2, which are then directed back to the J_1 states, as in Figure 3.12. Suppose, for example, that state s made a transition to s' with probability $\mathcal{T}_1(s,s')$ and discount β_1 in J_1 and probability $\mathcal{T}_2(s,s')$ and discount β_2 in J_2, then s_1, in copy 1, would make a transition to s'_2, in copy 2 with probability $\mathcal{T}_1(s,s')$ and discount β_1 and s_2, in copy 2, would make a transition to s'_1 in copy 1 with probability $\mathcal{T}_2(s,s')$ and discount β_2.

If Lemma 1 is used to remove the states added for the J_1 dynamics, the resulting model will be equivalent to (not just interchangeable with) J_3. Thus, operator concatenation has a direct interpretation as an operation on an SMDP state space.

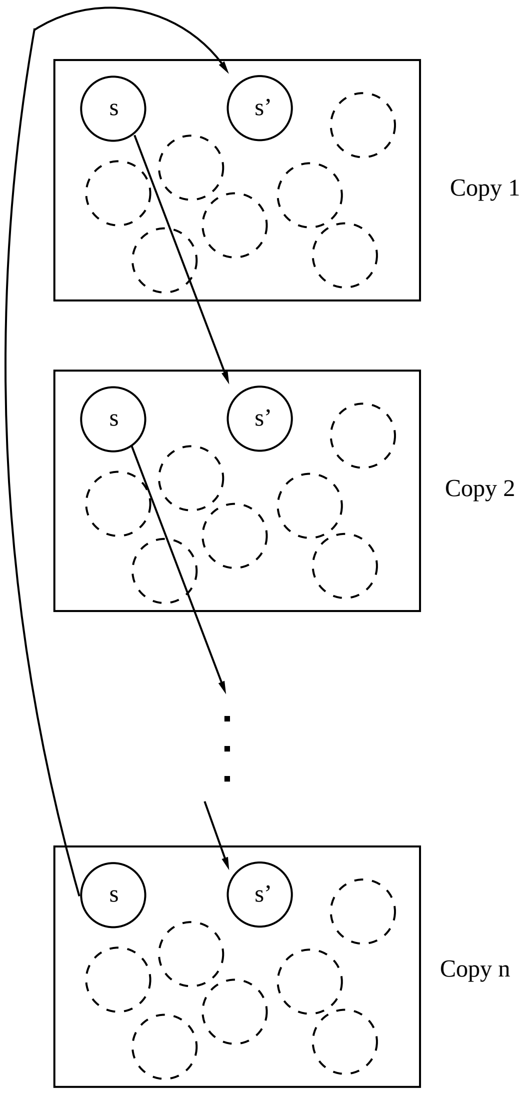

Figure 3.11: Model M has been unrolled n steps by making n copies and connecting them serially.

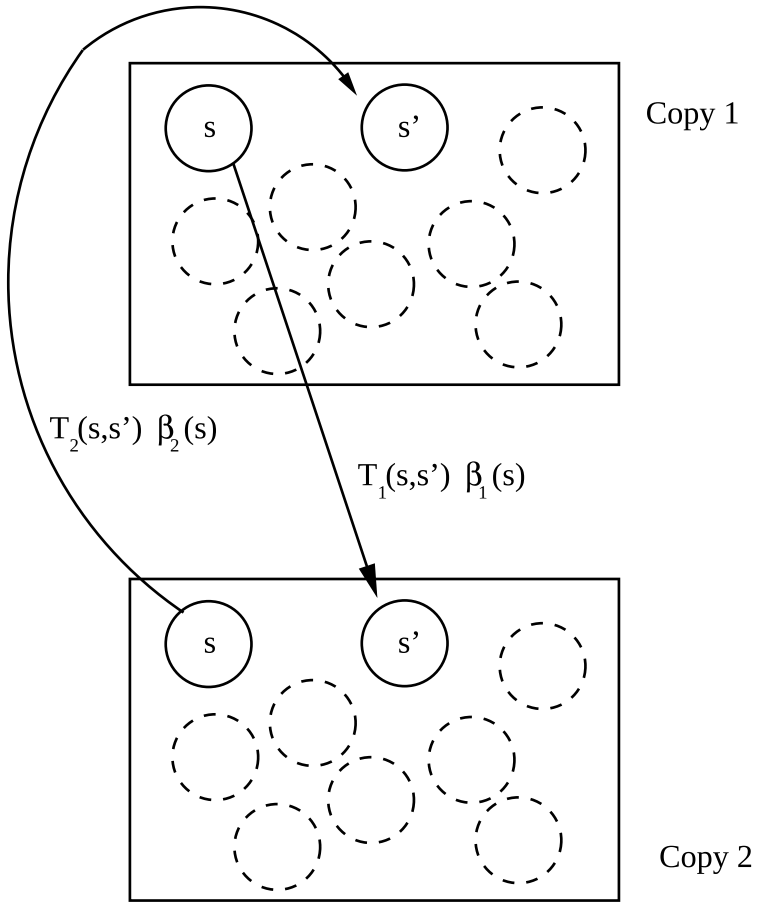

Figure 3.12: Two operators are composed by creating two copies of the state space and connecting according to the dynamics implied by the operators.

Averages of operators

The weighted average of a set of operators, $J_1 \dots J_i$, by some weights, $w_1 \dots w_i$ ($\sum_j w_j = 1$), is interpreted as the construction of new transition and reward functions so that

$$\mathcal{T}_{total}(s, s') = \sum_j w_j \mathcal{T}_j(s, s'),$$

and

$$\mathcal{R}_{total}(s, s') = \sum_j w_j \mathcal{R}_j(s, s').$$

Alternatively, this can be thought of as a stochastic operator J_A, which chooses J_i with probability w_i. The stochastic update will be interchangeable with any of the individual update rules because each of the J_i is a contraction with the same fixed point, so the order or relative frequency in which they are applied has no bearing on convergence.

There is also an interpretation of the averaging operation in terms of a manipulation of an SMDP. This is done by creating $i + 1$ copies of the states. The k^{th} copy of state s operates according to the reward function and transition function expressed in J_k, but makes transitions to the 0^{th} copy of the destination state. That is,

$$\mathcal{T}_A(s_k, s'_0) = \mathcal{T}_k(s, s').$$

The reward function is carried over directly,

$$\mathcal{R}_A(s_k) = \mathcal{R}_k(s)$$

as is the discount,

$$\beta_A(s_k) = \beta_k(s)$$

The 0^{th} version of state s flips a coin and moves to the k^{th} version with probability w_k:

$$\mathcal{T}_A(s_0, s_k) = w_k$$

The time required for transitions from 0 version states is 0, i.e. $\beta(s_0) = 1$ and the reward is also 0. [7]

[7]Discount factors of 1 are generally avoided here because they can cause difficulties in contraction proofs. In this case, the model is immediately transformed to an equivalent model with all discounts less then 1, so there is no cause for concern.

The intuitive interpretation of this is that for each state, the model instantaneously flips a biased i-sided coin and then proceeds with the version of the model dynamics corresponding to the result of the flip as in Figure 3.13. When every state except the 0 version states is removed (Lemma 1), the resulting model will have transition and reward functions that are weighted combinations of the original transition and reward functions and the backup operator will be equivalent to the average backup operator, J_{total}.

Pointwise combinations of operators

The convergence of pointwise combinations of the above operators follows as a straightforward generalization of their counterparts due to the convergence of asynchronous value determination. They will not be covered here beyond this observation.

Full β-models

A full β-model is a weighted multi-step operator. This combines the features of multi-step operators and averaged operators in a way that can assign different weights to different trajectories through the model. Each state can have a unique weighting factor, ω_s.[8] The weight attributed to step i in the model is

$$w = (1 - \omega_{s(i)}) \prod_{k=1}^{i-1} \omega_{s(k)},$$

where $s(i)$ is the state at time i.

The assignment of different weights to different states leads to the unusual property of full β-models by which different trajectories through the model can have different weights. For example, if $\omega_s = 0$, then any trajectories that pass through state s will have weight 0 after the first occurrence of s. However, states in the portion of the trajectory before s is reached may have non-zero weight. In general, instead of having a termination condition that specifies a fixed number of steps in a multi-step model, full β-models can be interpreted as assigning an exponentially decayng weight scheme to all possible multi-step trajectories through the model.

Full β-models have a surprisingly simple interpretation in the SMDP framework. They are expressed as a combination of the coin flipping gadget used to average operators

[8]Sutton used β, hence the name β-models. However, this would cause too much confusion with the use of β here as a discount factor.

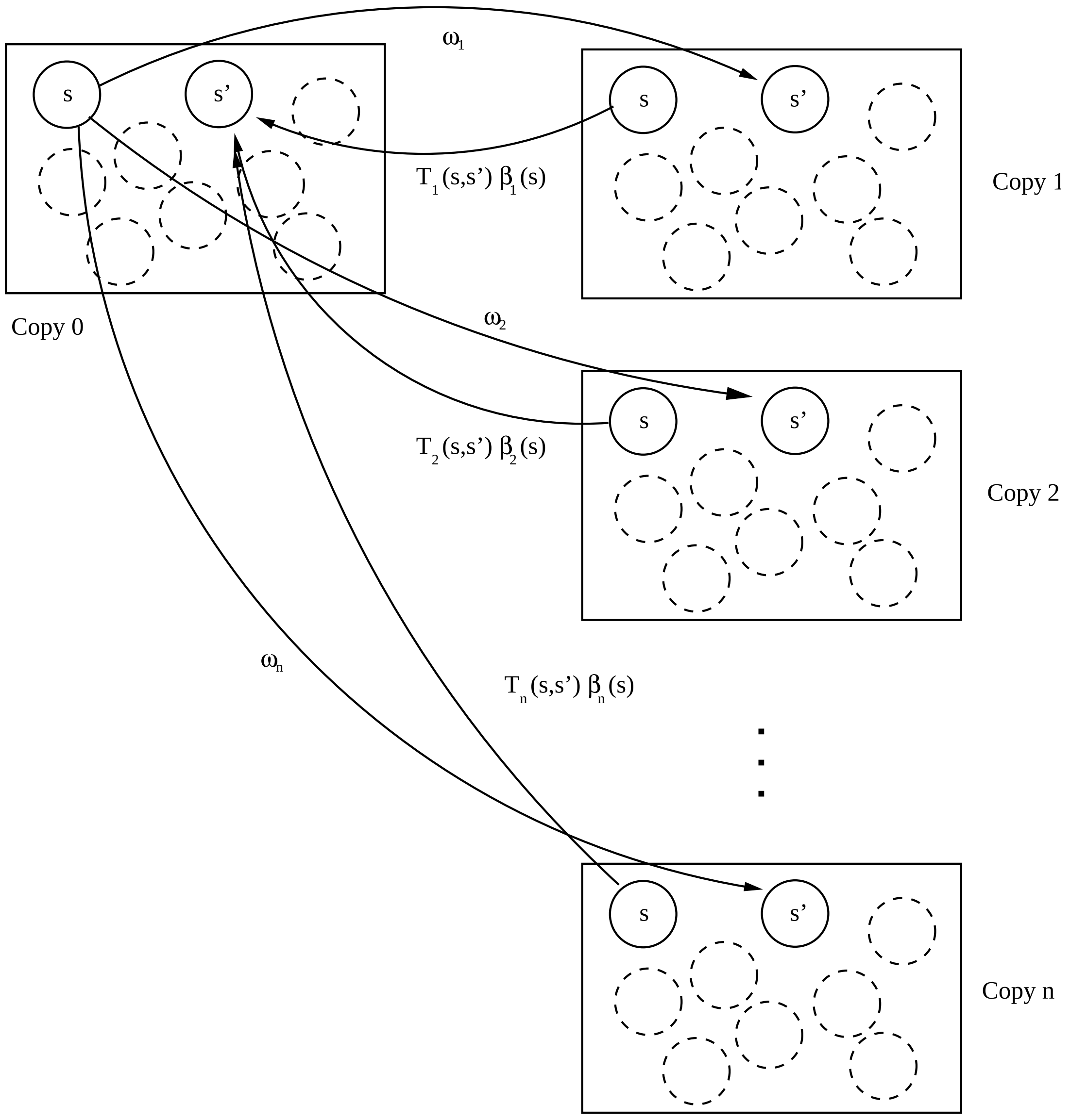

Figure 3.13: A coin flipping state that chooses stochastically between states $s_1 \ldots s_i$, which implement $J_1 \ldots J_i$.

and the unrolling trick used for multi-step operators. Starting with some original transition function, $\mathcal{T}$, and reward function, $\mathcal{R}$, two copies of the states in the original model are made. For states s and s', there are now states $s_1 \dots s_2$ and $s'_1 \dots s'_2$. The process stays inside the first version of the state space with probability $1 - \omega_s$. Thus, if the original state space had a transition $\mathcal{T}(s, s')$, the version 1 copies of the states obey:

$$\mathcal{T}(s_1, s'_1) = \mathcal{T}(s, s')(1 - \omega_s),$$

and

$$\mathcal{T}(s_1, s'_2) = \mathcal{T}(s, s')\omega_s$$

with

$$\mathcal{R}(s_1) = \mathcal{R}(s_2) = \mathcal{R}(s)$$

and

$$\beta(s_1) = \beta(s_2) = \beta(s)$$

The second version stays inside the second copy of the states with probability ω_s and returns to the first with probability ω_s:

$$\mathcal{T}_\beta(s_2, s'_1) = \mathcal{T}(s, s')(1 - \omega_s)$$

and

$$\mathcal{T}_\beta(s_2, s'_2) = \mathcal{T}(s, s')\omega_s.$$

This is shown conceptually in Figure 3.14. When all of the states except those in the first version are removed (Lemma 1), the resulting transition model will be equivalent to the β transition model that would result if the infinite summation over every possible trajectory were to be computed.

To see this, consider the interpretation of averaging as a stochastic DP operator. The weight assigned to a particular operator corresponds to the probability that this operator is applied. The number of steps corresponds to the number of coin flips before the model returns to version 1. Suppose that the process has gone through a trajectory, $s(1) \dots s(k)$, where $s(i)$ is the state at time i. Given this trajectory, the probability of returning to version 1 in exactly k steps is

$$(1 - \omega_{s(k)}) \prod_{i=1}^{k-1} \omega_{s(i)},$$

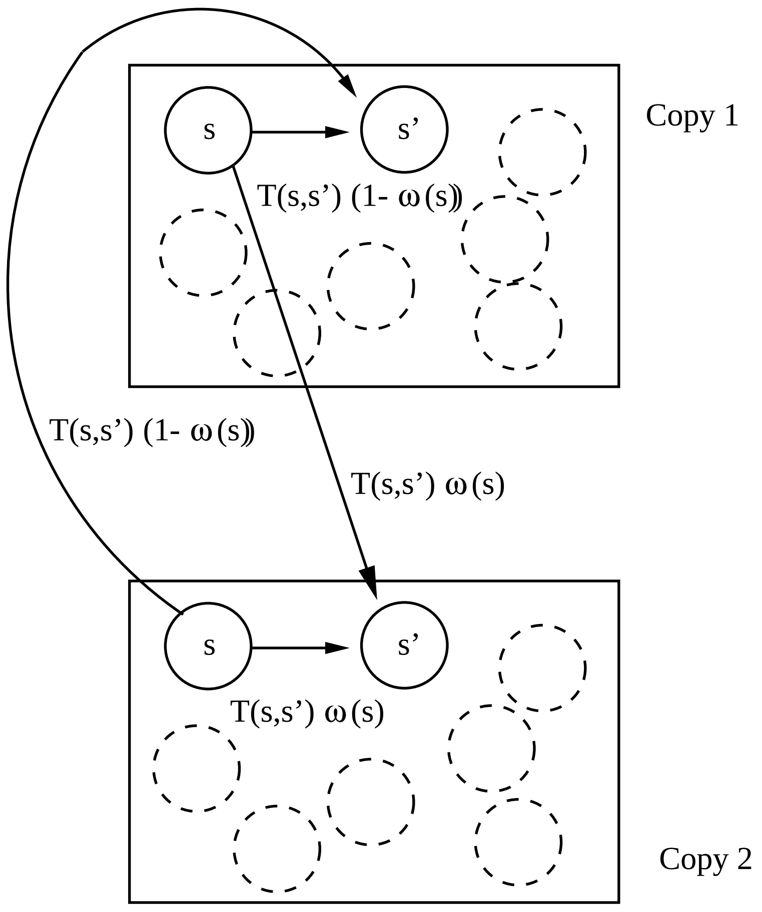

Figure 3.14: A transformation on the original model and transition function, $\mathcal{T}$, that is equivalent to a full β-model

which is exactly the weight assigned to this trajectory by the β-model. This makes the model a stochastic implementation of the exponentially decaying average implicit in β-models.

Note the similarity between this interpretation of full β-models and the unreliable robot actuation earlier in this chapter. In both cases, an infinite summation over time turns out to be equivalent to an SMDP model with some extra, uncontrolled states. This interpretation makes β-models easier to work with and makes them amenable to the algorithms of Chapter 4.

Macros and Options

Recent work by Precup, Sutton and Singh (Precup & Sutton, 1997; Sutton, Precup, & Singh, 1998) has considered the more general case where an agent may have a choice between executing several temporally extended policies of the form described above. The policies are used as abstract actions and are called *macros* (Precup & Sutton, 1997), in the spirit of macro operators in classical planning, or, more recently (Sutton et al., 1998) *options*.

Options are, essentially, a more formal version of S.-H. Lin's robot subtasks for reinforcement learning (Lin, 1993). While Lin specified fixed points at which subtasks terminated, options can use the exponentially decaying stopping conditions specified by full β-models. However, the overall effect is much the same. As has been shown above, options also have an interpretation as SMDP actions, and the convergence of value iteration, policy iteration or reinforcement learning with options all follow as a consequence of the convergence of these algorithms for SMDPs.

The authors of the option literature have advocated a use of temporal abstractions that is somewhat different from that advocated by other authors. Typically, temporal abstractions have been viewed as a *replacement* for low-level actions. Forestier and Varaiya (1978) and Dean and Lin (1995) both used temporal abstractions as a way of simplifying reasoning about the effects of policies over large regions of the state space. These applications of abstract actions can be called *reductive* since they aimed to use abstract actions as a means of reducing the size of the state space. L.-J. Lin did not aim explicitly to remove states from the state space, but did mention the need for reducing the number of points at which actions could be initiated or stopped, effectively reducing the size of the state space. In contrast, the advocated use of options is *concurrent* with low-level actions, placing no restrictions on which abstract or low-level actions can be initiated at any time.

The concurrent use of temporally abstract actions with low-level actions does have some advantages. It merely adds actions to the original MDP and since the actions correspond to actual policies over regions of the MDP, the optimal solution to the MDP will not change. This is in contrast to the Forestier and Varaiya approach, where the optimal policy for the simplified MDP may not be the optimal policy for the original MDP. However, the added flexibility of the option-style use of temporally abstract actions comes at a price. The new decision problem will have the same size state space and *more* actions, making each iteration of policy iteration or value iteration more costly, and potentially requiring more environment experiences in reinforcement learning. The advocates of options argue the use of options will speed convergence, requiring fewer iterations with off-line methods and less random roaming about in the environment in reinforcement learning. This may only be the case, however, if the temporally abstract actions are well-chosen. Recent arguments by Hauskrecht et al. (1998) indicate that abstract actions that move an agent away from the interesting parts of the state space actually can slow convergence.

3.5 Conclusion

This chapter presented the SMDP framework and showed that this framework captures the essence of a large number of approaches to temporal abstraction that have been adopted in the MDP field. It presented a new convergence proof for Q-learning for SMDPs and showed that the convergence and optimality properties of other temporal abstraction methods all follow as a consequence of SMDP convergence and optimality results. The following chapter discusses algorithms for converting temporally abstract actions into SMDP action descriptions.

Chapter 4

Symbolic Methods - Algorithms that Convert Policies to Actions

The previous chapter described the semi-Markov decision process framework and showed that a variety of temporal abstraction methods could be understood within this framework. A common mechanism in these methods is the step of fixing the policy in some region of the state space and then removing the states for which the policy is fixed. The act of executing the policy over the region then becomes a temporally abstract action that can be launched by the states bordering the region. The process of removing states to produce an abstract action was shown to be equivalent to a sequence of state removal operations performed on an SMDP.

In spite of this clear formal connection between SMDP actions and the act of executing a policy over a region of the state space, there has been some confusion in the AI community about mechanisms for efficiently converting a policy, or piece thereof, to an action. This chapter presents a new class of algorithms called *symbolic* methods, with on-line and off-line variations, that achieve this transformation, and a table is presented showing the advantages and disadvantages of each algorithm. The relationship between these algorithms and standard MDP algorithms also is demonstrated.

The representations and techniques used in some of the algorithms in this chapter suggest a new class of optimality criteria for MDPs. The chapter concludes with a generalization of MDPs called symbolic MDPs (SyMDPs). This generalization retains the formal properties of the MDP framework, but permits more interesting optimality criteria that can trade off rewards or costs with weighted combinations of probabilities of event outcomes at prespecified rates. This makes it possible to find policies that will choose an outcome only if its expected value is greater than some constant, or policies that will trade the likelihood of terminating in one state against the likelihood of terminating in another at some prespecified rate.

4.1 Off-line algorithms

This section describes off-line algorithms that convert policies defined over regions of a state space into SMDP action descriptions.

4.1.1 The one-at-a-time algorithm

The *one-at-a-time* algorithm follows directly from the proof of Theorem 7 (Chapter 3). The first step is to augment the state space of the original MDP adding uncontrolled states that execute policy π on region $G \subseteq \mathcal{S}$. This is done as follows: for each state, $s \in G$, add state s_1. For $s' \notin G$, define $\mathcal{T}(s_1, s'_1) = \mathcal{T}(s, \pi(s), s')$, so state s_1 mimics the behavior of state s under policy π. There is no action in the transition function of s_1 because s_1 is an uncontrolled state. For $s' \notin G$, define $\mathcal{T}(s_1, s') = \mathcal{T}(s, \pi(s), s')$, so the process returns to the controlled region upon leaving G. For each s in G from which π can be started, create a new action, a', such that $\mathcal{T}(s, a', s'_1) = \mathcal{T}(s, \pi(s), s')$ for all $s' \in G$ and $\mathcal{T}(s, a', s_1) = \mathcal{T}(s, \pi(s), s')$ for all $s' \notin G$. Thus, action a' shifts from the regular states in the MDP to a set of uncontrolled states that model the effects of policy π on G. In general, the termination conditions do not need to be this sharply defined. For example, using the structures developed in Section 3.4.4, stochastic termination conditions can be added to the region.

According to Lemma 1 from Chapter 3, the uncontrolled states can be removed from the process, creating an equivalent SMDP in which the act of launching π is summarized with a single SMDP action. Let function sremove(M,s) remove state s from MDP M as specified by Lemma 1 (This is not reproduced here in pseudocode because the equations are too big.) The following algorithm will remove all of the uncontrolled states:

```
function oaat-reduce ($M$)
  Let s = some uncontrolled state in M
  if $s = nil$ then
    return $M$
  else
    return oaat-reduce(sremove(M,s))
```

The worst case complexity of sremove is $O(|\mathcal{A}||\mathcal{S}|^2)$, which would occur if every state makes transitions to the removed state for every action, and if the removed state has transitions to every other state as well. This results in a worst case complexity of $O(j|\mathcal{A}||\mathcal{S}|^2)$ for oaat-reduce where j is the number of states removed. In the more realistic case, where the connectivity of the state space is bounded by k, the complexity is $O(j|\mathcal{A}|k^2)$, under the assumption that connectivity never grows above k while oaat-reduce is running. Note that while the connectivity of the model can indeed grow as states are removed, the total number of edges in the transition graph representing the MDP cannot increase. Thus, the cost of value determination for the model that results when the states are removed can be no greater than the cost would be if the states were left in the model. Unless every state inside of the region is connected to every state outside of the region, the cost will be lower. Whether this lower cost will amortize the expense of removing the states will depend upon other factors such as the topology of G, the number of times the abstract action is used by value iteration or policy iteration in the reduced model, and even the order in which the states are removed.

As a simple example of state removal, consider a room from the four-room navigation problem with the policy specified as shown in Figure 4.2. Suppose that the states are labeled as shown in Figure 4.1. For this example, assume the following transition model: Each action will move the agent in the intended direction with probability 0.7, and in one of the remaining axis-parallel directions with probability 0.1 for each. Bumping into walls has no effect. For this policy, $\mathcal{T}(s_{12}, s_{17}) = 0.7$, $\mathcal{T}(s_{12}, s_7) = \mathcal{T}(s_{12}, s_{13}) = \mathcal{T}(s_{12}, s_{11}) = 0.1$. When state s_{12} is removed from the model, the new transition probabilities for state s_7 will be $\mathcal{T}(s_7, s_2) = \mathcal{T}(s_7, s_6) = \mathcal{T}(s_7, s_8) = \mathcal{T}(s_7, s_{11}) = \mathcal{T}(s_7, s_{13}) = 0.1$, and $\mathcal{T}(s_7, s_{17}) = 0.5$. The discount factors are $\beta(s_7, s_2) = \beta(s_7, s_6) = \beta(s_7, s_8) = \beta$ and $\beta(s_7, s_{11}) = \beta(s_7, s_{13}) = \beta(s_7, s_{17}) = \beta^2$. A uniform discount factor can be obtained by normalizing the Bellman equation coefficients for s_7 to turn each coefficient into a probability. The normalizing factor

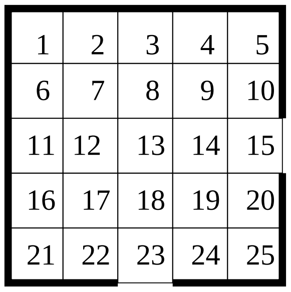

Figure 4.1: A labeling of the states in room 1.

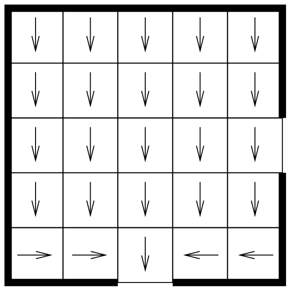

Figure 4.2: A policy for room 1.

becomes the new discount rate:

$$\begin{aligned}\beta' &= \sum_i \mathcal{T}(s_7, s_i)\beta(s_7, s_i) \\ &= 3(0.1\beta) + 2(0.1\beta^2) + 0.5\beta^2 \\ &= 0.3\beta + 0.7\beta^2.\end{aligned}$$

The new transition probabilities become $\mathcal{T}(s_7, s_2) = \mathcal{T}(s_7, s_6) = \mathcal{T}(s_7, s_8) = \frac{0.1\beta}{\beta'}$, $\mathcal{T}(s_7, s_{11}) = \mathcal{T}(s_7, s_{13})\frac{0.1\beta^2}{\beta'}$, and $\mathcal{T}(s_7, s_{17}) = \frac{0.5\beta^2}{\beta'}$.

Note that in this case, the connectivity of state s_7 has increased from 4 to 7. However, the total number of edges in the state transition graph for this model has decreased since state s_{12}, which also had a connectivity of 4, is now removed.

4.1.2 The all-at-once algorithm

The all-at-once algorithm works by directly manipulating the system of linear equations that defines the values of the uncontrolled states in terms of the values of the controlled states. The goal is to achieve a form that expresses the value of every uncontrolled state as a linear function of just the controlled states. As in ordinary value determination, the Bellman equation for each uncontrolled state becomes a row in a system of equations. However, the system is not square — there will be no rows for controlled states. If the states are sorted so that the controlled states are numbered higher than the controlled ones, the system can be expressed as the following partitioned matrix:

$$\left(\begin{array}{ccc|ccc|c} u_{11} & u_{12} & \dots & c_{11} & c_{12} & \dots & k_1 \\ u_{21} & u_{22} & \dots & c_{21} & c_{22} & \dots & k_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & k_3 \end{array}\right)$$

where row i comes from the Bellman equation of state i, u_{ij} comes from the coefficient of uncontrolled state j in terms of controlled state j, c_{ij} comes from the coefficient of uncontrolled state i in terms of uncontrolled state j, and k_i comes from the constant in the Bellman equation, i.e., the reward.

When this system is solved to determine the values of the uncontrolled states it produces the following form:

$$\left(\begin{array}{ccc|c} 1 & 0 & \dots & f_1 \\ 0 & 1 & \dots & f_2 \\ \vdots & \vdots & \ddots & \vdots \end{array}\right)$$

where the functions $f_1 \dots f_n$ are linear functions that express the values of the uncontrolled states in terms of the controlled states. For any state that executes an action that moves to state u_i, the expression $V(u_i)$ can be replaced in the Bellman equation with $f_i(V(c_1), \dots, V(c_j))$ if there are j uncontrolled states.

The complexity of removing a block of states by this method will depend upon several factors: the number of states in the uncontrolled region, the number of controlled states to which the uncontrolled region makes transitions, and the topology of the connections inside the region. If the uncontrolled part of the system can be arranged in a block-diagonal fashion, then this method can be quite efficient. Heuristics exist for rearranging "ill-conditioned" matrices to make them more amenable to elimination methods. See Lipton, Rose, and Tarjan (1979), which develops fundamental matrix conditioning algorithms that are still the basis of the current best heuristic methods.

In the four-room example, suppose that the two states in which the policy can terminate are the states in the adjoining rooms, labeled s_{26} and s_{27} as in Figure 4.3. States s_1 through s_{25} would form the left-most square block of the system of equations, and states

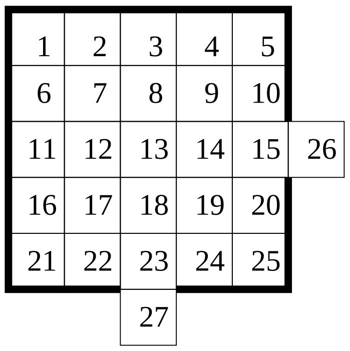

Figure 4.3: Terminating states s_{26} and s_{27} for policy π.

s_{26} and s_{27} would form a two-column strip between these states and the constant. The solution to this system yields a 5×5 identity matrix and a three-column strip that shows the value of each state in the room in terms of s_{26} and s_{27}. This is shown in Table 4.1 for the policy in Figure 4.2 for $\beta = 0.95$.

4.1.3 Symbolic value determination

The all-at-once algorithm of the previous section solves a system of linear equations directly to determine the value of the states in an uncontrolled region of an SMDP in terms of the values of the neighboring states. If this system of equations is too large, it may be necessary to use an indirect method, such as Gaussian iteration, to solve the system. The iterative value determination algorithm from Chapter 2 can be modified to fulfill this role. There are two ways to think of this modification. If there are j bordering, controlled states:

1. The value of controlled state s_i that borders the uncontrolled region is a token $\mathcal{V}_i$ that is propagated back through the Bellman equation for each state. The value of each state is then a symbolic expression in terms of these tokens and a constant.

2. The value of controlled state s_i is a vector, $\mathcal{V}$ with $\mathcal{V}[i] = 1$ and $\mathcal{V}[k] = 0$, for $k \neq i$, and the value of any uncontrolled state is a linear combination of these vectors.

These two representations are equivalent since they are just different ways of expressing linear functions. The vector representations turns out to be somewhat tidier: the "value" assigned to each uncontrolled state is a $j + 1$ dimensional vector, where the i^{th} position of the vector indicates the value as a function of controlled state i. The $j + 1^{st}$ position of the vector is used to carry reward information.

State	Value
1	$0.00V(s_{26}) + 0.60V(s_{27})$
2	$0.00V(s_{26}) + 0.61V(s_{27})$
3	$0.01V(s_{26}) + 0.63V(s_{27})$
4	$0.03V(s_{26}) + 0.60V(s_{27})$
5	$0.09V(s_{26}) + 0.54V(s_{27})$
6	$0.00V(s_{26}) + 0.64V(s_{27})$
7	$0.00V(s_{26}) + 0.65V(s_{27})$
8	$0.00V(s_{26}) + 0.67V(s_{27})$
9	$0.02V(s_{26}) + 0.64V(s_{27})$
10	$0.10V(s_{26}) + 0.57V(s_{27})$
11	$0.00V(s_{26}) + 0.68V(s_{27})$
12	$0.00V(s_{26}) + 0.71V(s_{27})$
13	$0.02V(s_{26}) + 0.74V(s_{27})$
14	$0.00V(s_{26}) + 0.70V(s_{27})$
15	$0.02V(s_{26}) + 0.60V(s_{27})$
16	$0.10V(s_{26}) + 0.73V(s_{27})$
17	$0.00V(s_{26}) + 0.77V(s_{27})$
18	$0.00V(s_{26}) + 0.81V(s_{27})$
19	$0.00V(s_{26}) + 0.76V(s_{27})$
20	$0.01V(s_{26}) + 0.72V(s_{27})$
21	$0.00V(s_{26}) + 0.78V(s_{27})$
22	$0.00V(s_{26}) + 0.83V(s_{27})$
23	$0.00V(s_{26}) + 0.90V(s_{27})$
24	$0.00V(s_{26}) + 0.83V(s_{27})$
25	$0.00V(s_{26}) + 0.78V(s_{27})$

Table 4.1: Values of states $1 \ldots 25$ in terms of $V(s_{26})$ and $V(s_{27})$. Coefficients are rounded to two decimal places.

In the room example from the previous subsection, s_{26} would have its value fixed at $[1, 0, 0]$ and s_{27} would have its value fixed at $[0, 1, 0]$. (The third position is reserved for rewards.) If the state values for this model are initialized to $[0, 0, 0]$, the first step of value determination would modify state s_{23} as follows:

$$\begin{aligned} V(s_{23}) &\leftarrow \beta[0.1V(s_{18}) + 0.1V(s_{22}) + 0.1V(s_{24}) + 0.7V(s_{27})] \\ &= \beta[(0, 0, 0) + (0, 0, 0) + (0, 0, 0) + (0, 1, 0)] \\ &= (0, \beta, 0) \end{aligned}$$

If there were a reward r associated with this step, than this reward would occupy the last position in the vector: $(0, \beta, r)$.

Symbolic value determination returns the same answer as the all-at-once algorithm:

Theorem 8 *Symbolic value determination for a region, G, of an MDP will converge to an assignment of vectors to states such that the i^{th} component of the vector value for state s equals the coefficient of the linear function for the value of state s in terms of state i. Furthermore, the value of the last component of each vector will be the expected, discounted, sum of rewards received upon proceeding from state s to the next controlled state.*

Proof: Each update in symbolic value determination performs $j + 1$ independent value determination steps in parallel. Each of these is individually a contraction at rate β. ■

The following is stated without proof and follows directly from the above theorem and the convergence of asynchronous value determination:

Theorem 9 *Asynchronous symbolic value determination will converge to the same answers as regular value determination under suitable non-starvation conditions.*

An important difference between symbolic value determination and the all-at-once algorithm is that symbolic value determination lends itself naturally to value function approximation. While value function approximation is not a focus of this dissertation, it is worth mentioning briefly how symbolic value determination could be combined with this method. Suppose that there is a large, uncontrolled region, G, that is to be removed from the state space. Since G is very large, it will be impractical to assign a value to every state in G and, instead, a function approximator will be used to estimate the value of the states in G based upon the values of some set of exemplars, $H \subset G$. Symbolic value determination takes on the usual form for this type of function approximation method, where $f(s, W)$ is the form of the function approximator and W is a set of parameters or weights determining the behavior of the approximator:

```
procedure approximate-value-determination
$W \leftarrow$ an initial (random) weight vector
Until termination do
  For all $s \in H$
    $V(s) \leftarrow R(s) + \beta \sum_{s'} T(s, s') f(s', W)$
  Fit $f(s, W)$ to $V(s)$
return($f(s', W)$)
```

Termination conditions for these types of algorithms are typically defined by a function of the maximum Bellman error at each of the states in H, i.e. the maximum difference between $V(s)$ at one iteration and $V(s)$ at the next iteration. What makes symbolic value determination different from ordinary function approximation methods is that $f(s', W)$ must return a vector. This can be achieved by using a neural network with several outputs, by using separate function approximators for each component of the vector, or by making the vector component an input to the function approximation method and cycling through different values of this input to construct the vector.

4.1.4 Non-contraction of Symbolic Value Iteration

It is natural to ask if symbolic value determination could be extended beyond simple value determination so that optimal policies could be determined at the same time as the symbolic representation of the value of following the policy. In Dean and Lin's iterative abstraction method (Dean & Lin, 1995) (see also Chapter 3), for example, new policies are generated for regions of the state space by determining the optimal policy under the assumption that states outside the region have a fixed, constant value. A method such as the all-at-once algorithm or symbolic value determination must then be applied before the new policy can be treated as an abstract action. Unfortunately, it appears that these two steps cannot be compressed into a single step in a straightforward way.

The natural combination of value iteration with symbolic value determination (symbolic value iteration) would maintain two value tables, a regular, scalar value table, as is typical with ordinary value iteration, and vector value table as is used for symbolic value determination. The update procedure for each state would work as follows:

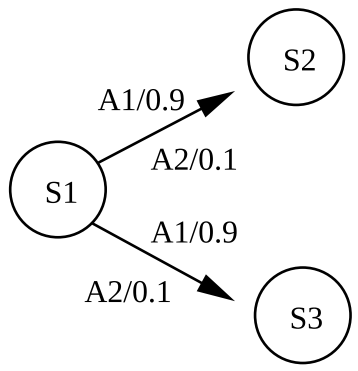

Figure 4.4: A model for which symbolic value iteration will not contract.

1. Update the scalar value table using value iteration
2. Update the vector value table using using the maximizing action from 1.

The problem with this is that the operation may no longer be a contraction in the maximum norm of the error in the coefficients. Consider the example in Figure 4.4, where the task is to determine the value of state s_1 in terms of the value of s_2 and s_3 at the same time a policy is determined for these states using value iteration. Suppose that the maximizing action for value iteration for state s_1 changes from a_1 to a_2, the symbolic representation of the value of s_1 would change from $(0.9\beta, 0.1\beta, 0)$ to $(0.1\beta, 0.9\beta, 0)$. In the course of value iteration, the maximizing action may change many times, causing the symbolic representation to flip-flop between these two possibilities. Clearly, this is not a contraction in the maximum norm of the coefficient errors as it was in the simple value determination case.

Once value iteration becomes fixed upon an optimal policy, this hybrid symbolic/scalar approach will converge to the correct symbolic representation. If there is more than one optimal policy, the maximizing action for value iteration may be unstable for quite some time. Still, it may be useful to use the hybrid approach before the policy is known to be stable if there is reason to believe that nearly optimal policies will have similar coefficient values to the optimal one.

4.1.5 Relationship to Asynchronous Dynamic Programming

There is a relationship between state removal and asynchronous policy iteration. Recall that for each phase of asynchronous policy iteration, a region, G, is defined, and one or both of the following operations are performed, in any order, for each s in G:

1. Update the policy, π, for state s.

2. Update the value for state s using action $\pi(s)$.

State removal algorithms operate on a region of the state space where the policy is fixed, producing a symbolic representation that instantaneously propagates value information across the removed states. Thus, every application of an abstract action is equivalent to an infinite number of value propagation steps in asynchronous dynamic programming, i.e., steps of the second type above. Moreover, an iterative abstraction algorithm such as the Dean and Lin algorithm can be interpreted as using two different phases of asynchronous policy iteration. Determining a new policy for region G under the assumption that the states bordering G have constant value is the same as performing an infinite number of both types of asynchronous policy iteration operators on the region G, while using the policy as an abstract action corresponds to an infinite number of value propagation steps with no policy improvement steps. This connection shows an alternate means of proving the convergence of iterative abstraction algorithms as a special case of asynchronous policy iteration.

4.2 Reinforcement Learning Methods

This section discusses on-line variants of the algorithms of the previous section. In this case, the problem is similar to the basic Temporal Difference (TD) learning problem (Sutton et al., 1998) (Chapter 2). In ordinary TD, the task is to learn the value of every state. In this section, the task is to learn the value of every state in some region, G, in terms of the states reachable from G in one step.

4.2.1 The Direct Method

It is worth pointing out that the simplest way to learn the value of any state under a fixed policy is to collect statistics. For any state, s, initialize the cumulative discounted reward, $\rho \leftarrow 0$ and and the cumulative discount, $\beta_c \leftarrow 1$. For each transition after s, that receives reward r:

$$\rho \leftarrow \rho + \beta_c r$$

and

$$\beta_c \leftarrow \beta\beta_c.$$

When a controlled state is reached, the destination is recorded, and these data are used for statistics on the discounted reward, the cumulative discount factor, and the transition probabilities between s and any controlled state bordering G. As shown in Chapter 3, these are sufficient to represent a transition from state s as an SMDP action.

This direct approach can be interpreted as a special case of TD(λ) and is called TD(1). This approach is not necessarily the most efficient in either case. Since TD(1) does not assign values to any of the intermediate states, there is no information shared across different trajectories through the model that share paths. Moreover, recent results by Sutton et al. (1998), described in more detail below, show that by learning the values of intermediate states, it is possible to remove states corresponding to several different abstract actions simultaneously.

4.2.2 Symbolic TD

One of the most important properties about the convergence proof for symbolic value determination (Theorem 8) is that it opens the door for symbolic temporal difference learning, (symbolic TD). As with symbolic value determination, if there are j states bordering region G, then the value function for every state is a $j+1$ dimensional vector. The value of state i that borders region G is a vector where component i is 1 and all other components are 0. A sampled reward, r, is treated as a vector, all components of which are 0, except the $j+1^{st}$, which is just r. As with the transformation from value determination to symbolic value determination, symbolic TD is implemented by replacing the regular TD operators with vector operations.

Theorem 10 *The Symbolic TD learning rule converges to true expression for $V(s)$ in terms of the states in C if*

1. *The state and action spaces are finite.*

2. $\sum_i \alpha_i(s,a) = \infty$ *and* $\sum_i \alpha_i^2(s,a) < \infty$ *uniformly over s and a w.p.1.*

3. *Var$\{r\}$ is finite.*

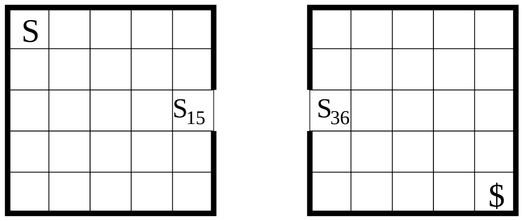

Figure 4.5: Solving the two-room navigation value determination problem as two separate symbolic TD problems.

4. $0 < \beta < 1$

Proof The proof mirrors the proof of the convergence of Q-learning for SMDPs since symbolic value determination is a contraction. ■

There are many potential applications of symbolic TD. It makes possible the resolution of "what if" scenarios. Suppose, for example, that a large system is under evaluation and the designer wishes to determine how an improvement in the overall system performance in one area depends upon another "critical" area. Symbolic TD could be used to determine the values of the states in the critical area in terms of the values of the states in the area that is under review. If the relationship between the areas is weak, i.e., the coefficients are small, this would indicate that a local improvement in the policy would not have a large effect on the critical area.

Several symbolic TD problems can be solved simultaneously and their solutions can be combined to produce the solution to the traditional, scalar TD problem. This approach may be faster than a direct solution to the scalar TD problem. Consider the two-room example, which has been split into two pieces in Figure 4.5. Given a policy for each of the rooms, the agent could use standard TD to find the value of each of the states. As with most algorithms based on dynamic programming, the time required to do this accurately will depend upon the "diameter" of the model, roughly the number of states through reward information must travel to be communicated to any other state, and the discount factor, which will bound the convergence rate. For this model, the diameter is roughly the width of two rooms.

Instead of directly learning the value of each state, the agent could learn the value of each state in the right room in terms of the value of s_{15}, and the agent could also learn the value of every state in the first room just in terms of state s_{36}. The agent might learn that value of state s_{15} is $0.85V(s_{36})$ and the value of state s_{36} is $0.1V(s_{15}) + 0.65$. This small system of equations could be solved to determine the values of s_{15} and s_{36}, which could then be substituted directly into the value functions for the other states. A key point here is that there is the potential for faster learning since the greatest distance information will travel is one room, not two as in the original, scalar TD learning problem. The tradeoff is that more work is required, in the form of solving a small system of equations, before a scalar value function can be produced. Another way to view this approach to temporal difference learning is as an exploration of a tradeoff between two extremes. Traditional TD, which directly learns a value function without learning anything like a transition model, is one extreme, while the other extreme would be a system that learns a complete transition model first and then solves for the state values off-line. This application of symbolic TD permits a tradeoff between the amount of computation that is done on-line in the form of TD updates, and the amount that is done off-line. The optimal point in this range will depend upon the relative cost of on-line vs. off-line computation as well as the form of the model. Models with "choke points" or small "interface" regions (Dean & Lin, 1995) (see also Chapter 6), such as the example here, will partition the state into easily decoupled regions and are good candidates since they will produce small systems of equations.

Another interesting property of symbolic TD is that when combined with the state space transformations of the previous chapter, it provides an alternative interpretation to the method introduced by Sutton (1995) for learning mixture models. A mixture model can be learned by symbolic TD by implicitly performing the state space manipulations described at the end of the previous chapter. Recall that these transformations involve augmenting the state space with several copies of the original states. While an agent operating in an environment does not have the option of augmenting the environment, the agent can simulate the effects of an augmented state space by maintaining internal state that tracks the agent's idea of which version of a particular state it is in.

For example, suppose that the agent is simulating a two-step model, which is implemented by unrolling the original model by one step and then connecting the second step back to the first (see Section 3.4.4). Thus, any sequence of states: s_1, s_2, s_3, s_4, where s_i is a random variable for the state at time i, would be interpreted by the agent as $s_1(1), s_2(2), s_1(1), s_2(2)$, where the parenthesized number denotes the alternation between the original and the unrolled version of the state space. By maintaining internal state information, the agent is able to simulate the effects of acting in the augmented state space

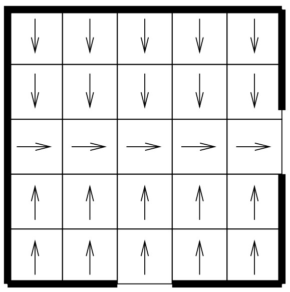

Figure 4.6: An alternate policy for the same room.

while still acting in the original one, constructing the unrolled state space from Chapter 3 in its “mind”. This effect works for transformations with coin-flipping gadgets as well, since the agent can internally flip a coin, implementing a stochastic transition between internal states.

A further benefit of the Symbolic TD approach is that it is amenable to an approach recently suggested by Sutton et al. (1998) for simultaneously learning the value of several different abstract actions based upon a single set of environment experiences. Consider the policy shown in Figure 4.6. Suppose that an agent is trying to use symbolic TD to learn vector state values for both this policy and the policy of Figure 4.2. The agent would need to alternate between different policies, updating vector state values for two different copies of the state space. This alternation may not be avoidable, but it is possible to share information between the two policies. Notice that while the policies produce significantly different behaviors, they assign the same action to many states. Thus, whenever the agent is in the top two rows of the model, any (state, reward, next-state) triple can be treated as if it occurred in both copies simultaneously and a step of symbolic value determination can be performed for both copies of the state space simultaneously. This observation can be seen as a generalization of a result in Harada (1997) in which an agent used a single experience to simultaneously update Q-values for different time slices in a finite horizon MDP.

Finally, it should be noted that as in symbolic value determination, function approximation can be used to approximate the coefficients in symbolic TD, with all of the standard pitfalls, benefits, and caveats.

4.2.3 Avoiding state elimination

So far, this chapter has focused on algorithms for removing states from the state space under the assumption that it is always useful to remove uncontrolled states. However, there is at least one case where it may may not be desirable to remove uncontrolled states. This situation arises in Q-learning, where the agent avoids explicitly constructing a model of the environment. In this case, the agent would like to learn the value of launching an abstract action directly, without incurring the overhead of symbolic TD. This can be done by applying the TD(1) version of symbolic TD. The agent initializes $\rho \leftarrow 0$ and $\beta_c \leftarrow 1$. For each transition after s, that receives reward r:

$$\rho \leftarrow \rho + \beta_c r$$

and

$$\beta_c \leftarrow \beta\beta_c.$$

When the agent reaches a controlled state, s', it performs a Q-learning update on the transition from s to s', treating the policy used over the region of the space as an action, treating ρ as the reward, and treating β_c as the discount factor. This must converge to the optimal policy since ρ and β_c are the true discount and reward for this trajectory in an SMDP and Q-learning was proven to converge for SMDPs in Chapter 3.

This shows that it is not *necessary* for a Q-learning agent to maintain state value estimates for the values of uncontrolled states. However, it could still be useful for the agent to do so for the reasons suggested in the previous subsection: faster convergence, and the sharing of information gained from a single experience across several different actions.

4.3 A Summary of State Removal Algorithms

This section presents Table 4.3, summarizing the properties of the algorithms presented so far in this chapter. They are categorized along several dimensions. The async. column indicates if the algorithm must be used synchronously or if it permits asynchronous versions. The func. approx. column indicates if the method is compatible with function approximation. The prob. type column indicates, in informal terms, the types of problems for which each method will excel. The on-line column indicates whether the method can be applied for reinforcement learning.

Algorithm	Async.	Func. Approx.	Prob. Type	On-line
one-at-at-time	Yes	No	Small	No
all-at-once	No	No	Block-diagonal	No
symbolic value determination	Yes	Yes	Large	No
symbolic TD	Yes	Yes	Large	Yes

Table 4.2: Features of the different algorithms in this chapter.

4.4 Crafting Actions — New Optimality Criteria

A negative result in this chapter is the example showing that the simultaneous determination of an optimal policy and a symbolic representation of this policy is not possible in a straightforward manner. In spite of the instability in the symbolic value representations when the maximizing action at a particular state changes, there is some underlying stability in symbolic value determination which *is* maintained when the maximizing action changes. This stability can be captured with a new type of optimality criterion, permitting specification of different types of policies that make explicit tradeoffs between the discounted likelihood of reaching various states. This class of optimality criteria does not appear to have been investigated before in the MDP community.

Consider again a single room as in Figure 4.3. If the values of states s_{26} and s_{27} are known, then the optimal policy for this room can be found by solving an MDP that contains just the states shown in the figure and has the values of s_{26} and s_{27} fixed at their correct values. If nothing is known about the value of s_{26} and s_{27}, some guess can be made, as is done in the initial phase of the Dean and Lin approach (Dean & Lin, 1995). In some cases, however, it is more natural to devise an abstract property that the policy must satisfy than it is to produce a sensible guess about a state value. For example, it might be worthwhile to start off with two policies, one that maximizes the expected, discounted probability of reaching state s_{27} and one that maximizes the expected, discounted probability of reaching state s_{26}. More interesting options may be desirable if some heuristic information is available about the region. For example, it might be desirable to find the policy that maximizes the expected, discount sum of the rewards received inside the room plus one half expected utility received for exiting through state s_{26}.

A decision problem that achieves this type of abstract requirement on a policy can be constructed using *symbolic optimality criteria.* A symbolic optimality criterion identities a set of terminal states, and a function of the expected, discounted probability of reaching these states and the expected, discounted sum of rewards.

A decision problem with a symbolic optimality criterion is called a a *symbolic MDP* (SyMDP), which can be defined as a 6-tuple, $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, f, \mathcal{C})$, where $\mathcal{C}$ is a set of boundary states in terms of which the other state values will be defined, and f is a function from vectors of size $|\mathcal{C}|+1$ to reals. The value for each state is a vector of size $|\mathcal{C}|+1$ and the Bellman equation for state $s \notin \mathcal{C}$ is:

$$V(s) = fmax_a \mathcal{R}(s,a) + \beta \sum_{s'} \mathcal{T}(s,s')V(s')$$

where $fmax_a$ is the value of the maximizing vector in $\max_a f(\cdot)$ and $\mathcal{R}(s,a)$ is a vector with component $|\mathcal{C}|+1$ equal to the reward for taking action a in state s. For s_i, the i^{th} state in $\mathcal{C}$, $V(s_i)$ is a unit vector with the i^{th} component equal to 1. The dynamic programming operator, J, is defined as:

$$V^t = J(V^{t-1}) = fmax_a(\mathcal{R}(s,a) + \beta \sum_{s'} \mathcal{T}(s,s')V^{t-1}(s')).$$

In Figure 4.3, a symbolic MDP that maximized the expected, discounted sum of rewards inside the room plus one half the expected, discounted probability of reaching state s_{26} would specify the set of boundary states as $\mathcal{C} = \{s_{26}, s_{27}\}$. The value of state s_{26} would be fixed at $[1,0,0]$, and the value of state s_{27} would be fixed at $[0,1,0]$. The value of each non-boundary state would also be a vector and the function, f, implementing the optimality criterion would be a function from state vector values, $\mathcal{V}$, to scalars. In this case it would be $0.5\mathcal{V}[1] + 0.5\mathcal{V}[3]$.

Theorem 11 *If f is a linear function, then the value iteration operator for SyMDPs is a contraction in the maximum norm of f applied to the individual state values.*

Proof: Consider first the case of a fixed policy. Suppose the maximum distance between V_x and V_y is ϵ, i.e., $\max_s |f(V_x(s)) - f(V_y(s))| = \epsilon$. Consider an update to V_y at state s under conditions that would maximize the difference between V_x and V_y at s: all successors of s differ by ϵ as well. For each state, construct an alternative representation of V_x such that $V_x(s) = V_y(s) + \mathcal{V}_\epsilon^s$. For all s, an appropriate $\mathcal{V}_\epsilon^s$ vector such that $f(\mathcal{V}_\epsilon^s) = f(V_x(s)) - f(V_y(s))$ must exist because f is linear. V_x is updated at state s as follows:

$$V_x^t(s) \leftarrow \mathcal{R}(s) + \beta \sum_{s'} \mathcal{T}(s,s')[\mathcal{V}_y^{t-1}(s') + \mathcal{V}_\epsilon^{s'}].$$

The update for V_y is:

$$V_y^t(s) \leftarrow \mathcal{R}(s) + \beta \sum_{s'} \mathcal{T}(s,s')[\mathcal{V}_y^{t-1}(s')],$$

and by the linearity of f:

$$\begin{aligned} |f(V_x^{t+1}(s)) - f(V_y^{t+1}(s))| &\leq f(\beta \sum_{s'} \mathcal{T}(s,s')\mathcal{V}_\epsilon^{s'}) \\ &= \beta \sum_{s'} \mathcal{T}(s,s') f(\mathcal{V}_\epsilon^{s'}) \\ &\leq \beta \sum_{s'} \mathcal{T}(s,s')\epsilon \\ &= \beta\epsilon \end{aligned}$$

What remains to be shown is that update is a contraction when the action changes. The above argument shows that for any state, s, $|f(Q_x^t(s,a)) - f(Q_y^t(s,a))| < \beta\epsilon$. Suppose that a_1 is maximizing for V_x and a_2 is maximizing for V_y, then

$$\begin{aligned} f(V_x^t(s)) &= f(Q_x^t(s,a_1)) \geq f(Q_x^t(s,a_2)) \\ f(V_y^t(s)) &= f(Q_y^t(s,a_2)) \geq f(Q_y^t(s,a_1)) \end{aligned}$$

Assume, without loss of generality, that $f(V_x^t(s)) \geq f(V_y^t(s))$, then since $f(V_y^t(s)) \geq f(Q_y^t(s,a_1))$, and $f(V_x^t(s)) - f(Q_y^t(s,a_1)) \leq \beta\epsilon$, $f(V_y^t(s)) - f(V_x^t(s)) \leq \beta\epsilon$. $\blacksquare$

A choice of f that returns the i^{th} component of the value vector and ignores all others will produce a policy that maximizes the discounted probability of reaching the i^{th} state in $\mathcal{C}$. If f ignores all but the $|\mathcal{C}| + 1^{st}$ component of the value vector, then the optimal policy will maximize the rewards received inside of the region. A choice of f that assigns the same coefficient to the i^{th} component of the value vector and the last component will trade rewards inside of the region at a rate of 1 to 1 against the value of the i^{th} state in $\mathcal{C}$.

Some of the effects of these optimality criteria could be achieved using scalar methods by making a clever guess about the values of the states in $\mathcal{C}$ and fixing the values at this guess. For example, if there are no rewards in the SyMDP then the policy that maximizes the discounted probability of reaching the i^{th} state in $\mathcal{C}$ is the same as the optimal policy for the region when the value of the i^{th} state is fixed at 1 and the values of all other states are fixed at 0. There does not appear to be an obvious way to guess state values to induce a specific tradeoff between internal rewards and bordering state values.

The following claims are verified easily using the techniques described in this chapter and are stated without proof:

1. SyMDPs with linear f can be solved using policy iteration.
2. SyMDPs with linear f can be solved using linear programming.
3. SyMDPs with linear f can be solved using Q-learning (with the standard assumptions).
4. SyMDPs with linear f can be generalized to Symbolic semi-Markov decision processes (SySMDPs) as can the theorems and results for SMDPs discussed in the previous chapter.

These results are related to an observation made in Szepesvári and Littman (1996) that the max in the Bellman equation can be replaced by any non-expansion. However, the scope here is somewhat different since the state values are vectors, not scalars, and linear functions can be expansions. The behavior of SyMDPs for non-linear symbolic optimality criteria is an open question.

4.5 Conclusion

This chapter presented several new algorithms for transforming policies defined over regions of a state space into SMDP actions. This is done by treating the states in which the policy is executing as uncontrolled states, and then removing the uncontrolled states from the state space. These algorithms are the nuts and bolts of the temporal abstraction methods described in the previous chapter and will be used heavily in the following two chapters.

The investigation of symbolic methods in this chapter also led to the consideration of some new optimality criteria for MDPs. These optimality criteria permit explicit tradeoffs between immediate rewards and the likelihood of reaching certain states. This chapter proved that when the tradeoffs in the optimality criterion are linear, stable dynamic programming methods can be used to find the maximizing policy.

Chapter 5

Hierarchies of Abstract Machines

5.1 Introduction

Chapter 3 presented temporal abstraction as a transformation of a policy defined over a region of state space into an action in an SMDP. A large number of temporal abstraction methods were then shown to be instances of this general approach. Chapter 4 presented several algorithms for performing the transformations described in Chapter 3. This chapter uses the tools that were developed in the previous chapters to develop a new, hierarchical approach to temporal abstraction in MDPs called Hierarchies of Abstract Machines (HAMs). The most important features of this approach are:

- An agent-centered abstraction language
- The hierarchical use of prior knowledge
- The optimal refinement of incompletely specified behaviors
- State space reduction

These features are realized within the temporal abstraction framework, making strong optimality and convergence guarantees possible, while also providing a link between temporal abstraction and behavior-based control (Brooks, 1986) or teleo-reactive control (Nilsson, 1994). The hierarchal nature of the HAM language also suggests connections with hierarchical approaches to classical planning (Tate, 1977).

This chapter also includes experimental results on a large navigation MDP with several thousand states. These results show that a small amount of fairly generic navigation knowledge expressed as a HAM can reduce dramatically the time required to produce nearly optimal policies.

5.2 Goals of HAM the method

HAMs were devised to overcome some limitations with existing abstraction methods, particularly temporal abstraction methods. While the results in Forestier and Varaiya (1978) showed that policy fragments could be combined optimally, they did not address the question of how such policy fragments could be devised in the first place. The example of automating the selection of different control programs for a factory made sense, since factories, particularly those running in 1978, usually ran a variety of human-designed control programs anyway.

The problems faced by Artificial Intelligence are somewhat different. Some knowledge about the right way to act in domains may be available, but is not usually in a specific form that clearly specifies actions for large chunks of state space. In fact, it is the non-specificity of human knowledge that makes it so powerful. Human beings appear to apply generic rules of thumb that don't match any *particular* situation, but that are quickly adapted to new challenges. For example, human beings have strategies for parking cars, moving around in buildings, cleaning houses, and mowing lawns. These strategies work for just about any car, parking lot, building or lawn. When a new situation is encountered, the generic strategy is modified with only a small fraction of the effort that would be required to achieve the new task with no prior knowledge. A description language for abstract actions should have this kind of generic structure.

Some artificial intelligence methods for temporal abstraction do incorporate prior knowledge. For example, Lin's robot subtasks (Lin, 1993), several avenues explored by Singh (1992) and recent work on "macros" and "options" (Precup & Sutton, 1997; Sutton et al., 1998) all construct abstract actions by making a guess that a particular subset of the states will constitute achievement of a subtask that is in some way relevant to the overall task. In these approaches, an abstract action is constructed by guessing a value for the subtask states and solving an MDP for which the subtask states are absorbing and have their values fixed at the guessed value (see Chapter 3). This approach can require a lot of domain-specific knowledge and guesswork about subtasks and subtask values.

The HAM approach takes a different strategy for producing complex behavior. This approach is rooted in the observation that engineers and control theorists are generally quite good at designing controllers that will realize specific low-level behaviors. A worthy goal for artificial intelligence should not be the mere duplication of these efforts using different means, but should be the leveraging of these accomplishments to achieve more interesting, higher-level tasks. Just as simple joint movements in the body are controlled by low-level feedback mechanisms in the nervous system, primitive robot operations should be handled using control theory techniques or highly specialized AI techniques. High-level functions that determine *when* or *why* to move a joint should be the purview of higher brain functions and of more general artificial intelligence methods. Unlike other approaches with externally imposed, problem-specific knowledge, HAMs use partially specified procedural knowledge, combined with an agent's sensory information about the environment, to induce a hierarchy of temporally abstract action choices. This permits a transformation of the original MDP to a reduced SMDP, the solution of which optimally refines an agent's generic knowledge, much in the same way a human being refines an abstract strategy to fit a specific situation.

5.3 Machine Policies

This section describes the use of machines as a means of specifying policies. In Chapters 3 and 4 policies were assumed to be mappings from states to actions and an abstract action corresponded to the execution of this mapping over a region of state space. This is an extremely clumsy representation. It is large, requiring space proportional to the size of the region on which the policy is defined. It is also extremely problem specific — a simple renumbering of the states will result in completely different behavior. These factors make it an awkward medium for the transfer of knowledge.

A cornerstone of the HAM approach is the idea that policies should be thought of as *programs* that produce actions as some function of the agent's sensor information. In the most general sense, these programs can be Turing machines that execute any computable mapping of the agent's complete sensor history to actions. This chapter focuses on stochastic finite state automata, which make the analysis of HAMs more tractable. The analysis of more general policy representations is reserved for future work (Chapter 7).

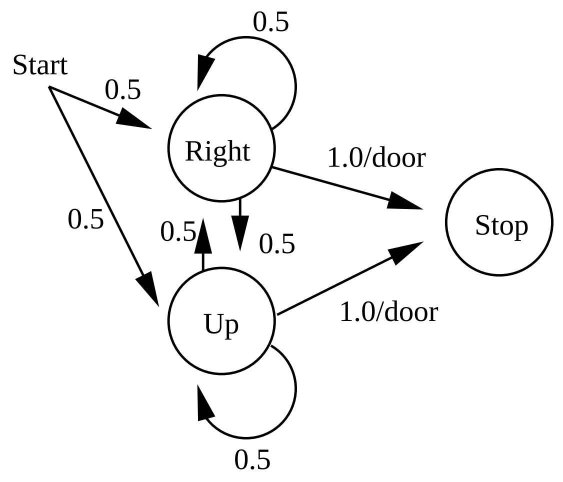

Figure 5.1: A simple Moore machine policy that moves right with probability 0.5, and up with probability 0.5, but stops if the agent has passed through a door. (Arcs are labeled as *probability*/*variable*, where *variable* can be thought of as a state or observation variable. If there is no variable label for an arc, this matches any variable not already mentioned. In this case, arcs labeled "0.5" match anything other than "door".)

A stochastic, finite state policy is essentially a Moore machine, with (possibly) stochastic transitions defined on some aspect of the state description. The outputs of the Moore machine correspond to actions. The machine may contain a distinguished *stop* state at which the action terminates. (These concepts will be defined more precisely when HAMs are defined formally in the subsequent section.) A simple, stochastic, Moore machine is shown in Figure 5.1. This machine moves up with probability 0.5 and right with probability 0.5. It stops when it passes through a door.

An agent executes a finite state policy by selecting the action associated with the agent's machine state output, observing the next environment state (or some information about the next environment state), then executing a machine transition based upon this information. This representation of a policy is "agent-centered" in the sense that it is defined in terms of the agent's sensory inputs. Specifically, the policy in Figure 5.1 assumes that the agent has some way of knowing when it has passed through a door. This type of description is necessary for policies that are meant to be generic objects usable in any MDP. Note, however, that this does not mean that the problem has changed to a incomplete information problem, or partially observable MDP (POMDP). It means simply that the agent is using a generic policy representation.

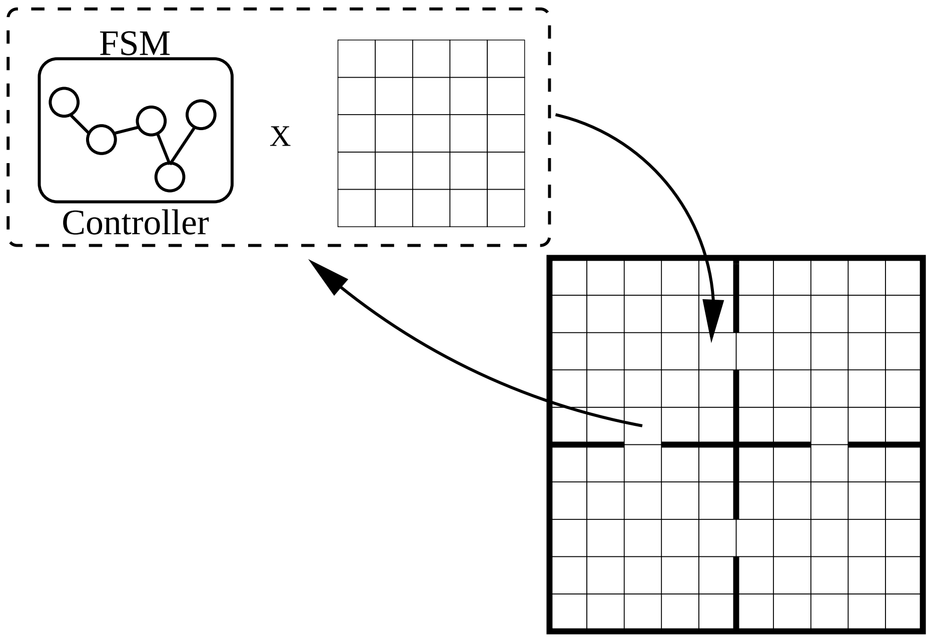

Figure 5.2: A finite state machine policy (FSM) is launched from the entrance to room 1. The act of executing this policy in room 1 is modeled as a stochastic process with states that are the cross-product of the environment states and the machine states. The FSM terminates when the agent reaches the entrance to room 2, and control returns to the original problem states.

The theorems of Chapter 3 and algorithms of Chapter 4 can be generalized to apply to policies that are expressed as stochastic, finite state Moore machines. The formal steps are described in detail in the succeeding section, but the basic idea is that, as in Chapter 4, a copy of that state space can be created for the region where the policy will be executed. However, instead of an exact copy, a cross-product of the environment states and the controller states is used to model the combined agent-environment system as a stochastic process (Figure 5.2). These cross-product states can then be removed from the state space using any of the algorithms from Chapter 4.

Note that this representation subsumes any of the abstract action representations discussed in Chapter 3. A simple mapping from states to actions can be achieved by creating one machine state for every action, making the explicit state number an observation, and encoding the mapping from states to actions in the machine state transition function. If the policy terminates in a particular set of states, this can be encoded as transitions to the machine's stop state. The stochastic stopping conditions of a full β-model (Sutton, 1995)

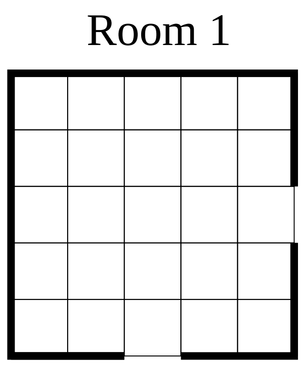

Figure 5.3: Room 1 of the four-room navigation problem.

(see Chapter 3) can be implemented as stochastic transition to the stop state. Multi-step and averaging operations are also implemented easily by applying the transformations of Section 3.4.4 to the machine states.

5.4 Hierarchical Machines

Suppose the policy of Figure 5.1 is applied starting at the bottom entrance of room 1 (Figure 5.3). This policy will bring the agent towards the right exit, but it may fail, overshooting the exit and driving the agent into the top right corner of the room. At this point, it might be better to have the agent stop and execute a different policy. If the agent is a robot with one sonar in each of the four coordinate directions, s-up, s-right, s-down, and s-left, then the the policy of Figure 5.1 could be modified to stop when the agent hits the top of the room as in Figure 5.4. Call this machine right-up. Another that moves right and down as in Figure 5.5 would be more appropriate in this case. Call this machine right-down.

A machine is *hierarchical* if it calls other machines as subroutines. The machine in Figure 5.6, uses subroutines by first calling the right-up machine, then calling the right-down machine only if the right-up machine fails. The control flow for a subroutine call is the same as that of an action. When a machine state with a subroutine call is entered, control is transferred to the starting state of the called machine. When the called machine reaches a stop state, control returns to the caller, which examines the current environment state to determine the next machine state.

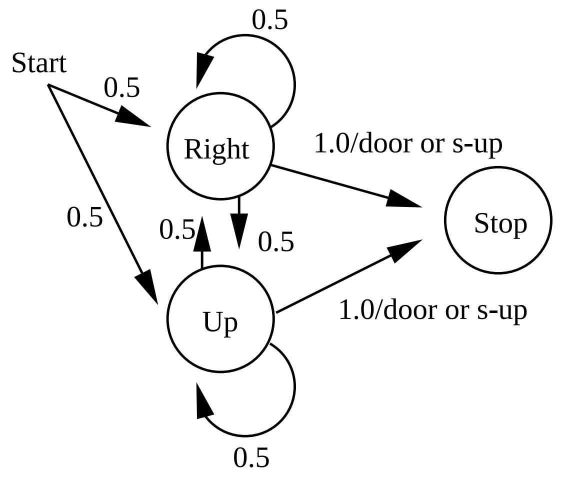

Figure 5.4: The right-up machine stops when sonar indicates that the top of the room is reached.

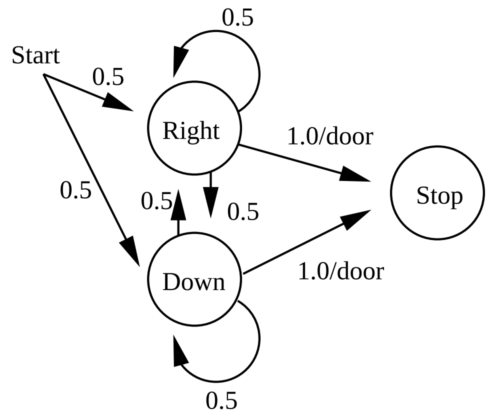

Figure 5.5: The right-down machine alternates between right and down until the agent passes through a door.

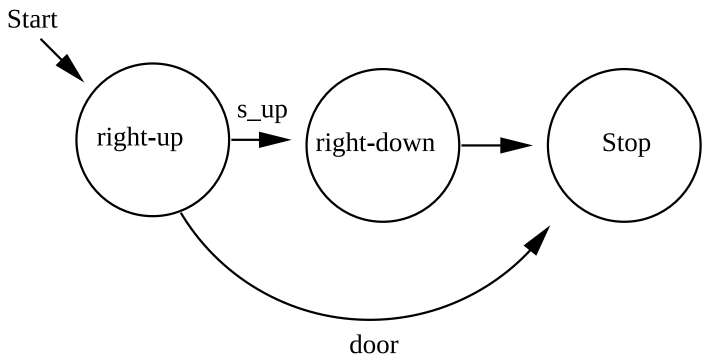

Figure 5.6: A machine that tries right-up first, then tries right-down if this fails. (Arcs without probability labels have probability 1.0.)

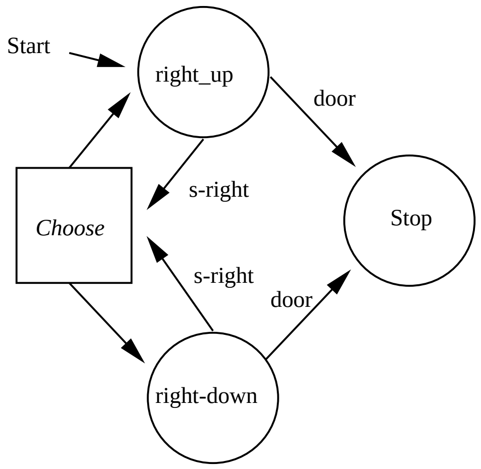

Figure 5.7: A machine with choice states.

5.4.1 Abstract Machines

The final feature that this chapter adds to finite state automata is by far the most interesting: the introduction of non-deterministic *choice* states, which make an automaton *abstract*, or partially specified. If actions in the environment have somewhat noisy outcomes, right-up and right-down won't do an especially reliable job of moving an agent from the bottom of room 1 to the right exit, even if the machines are chained together as in Figure 5.6. When the agent's up-pointing sonar s-up is active, it is clear that right-down is the better of the two machines. If the agent's s-down sonar is active, then right-up is clearly better. However, if only s-right is active, then the agent is at the right wall. This means that the agent has missed the door and that it might be a good time to switch policies. The machine in Figure 5.7 uses a choice state to specify that a choice about which machine to execute next must be made at this point. It indicates that one of the possible next states should be chosen, but does not commit to which.

The construction of the machine in Figure 5.7 assumes that right-up and right-down machines have been modified so that they stop when the right sonar is active. This machine will enter a choice machine state at the environment states indicated in Figure 5.8. These are the states at which the behavior is incompletely specified. By making the right choice at each of these points, the effectiveness of the two extremely simple subroutines is enhanced greatly. The machine in Figure 5.7 describes an abstract plan for moving from the bottom entrance to the right exit in a room. It does not depend on the size of the room at all, yet

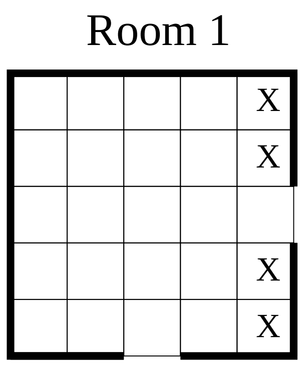

Figure 5.8: Environment states that cause machine choice states are marked with an X.

it identifies key points at which the plan must be refined to improve its effectiveness for a particular room. Of course, this is just a simple, informal example, and there are many obvious ways that the machines could be improved and made more or less general.

The following sections will formally define HAMs as the class of machines implementing the types of machine policies described in this and the preceding two sections. These sections will also show that the problem of optimally refining a HAM for particular environment can be interpreted as an SMDP.

5.5 HAM definitions

A *machine* for a HAM is a triple $\mathcal{N} = (\mu, \mathcal{I}, \delta)$, where μ is a finite set of machine states, $\mathcal{I}$ is a stochastic function from environment states to machine states that determines the initial machine state, and δ is a stochastic next-state function mapping from machine states and environment states to next machine states. For generality, $\mathcal{I}$ and δ will typically be a function of some state variables describing the environment state. There are three types of machine states:

action states specify an action to be taken in the current environment state. For an action state m, $\pi(m)$ is the action specified by m.

call states execute another machine as a subroutine. For call state m, $\pi(m)$ identifies the called machine.

choice states nondeterministically select a next machine state.

stop states halt execution of the machine.

Action states are the only states that execute actions in the environment. The transition function is not defined for stop states. For choice states, δ returns a list of possible next states.

The execution rules for a machine are best described in pseudo-code. In the program below, environment(a) is a function that executes action a and returns the next environment state. The actual environment state is assumed to be a local variable maintained by the environment function. The function type(m) returns the type of a machine state m. The function choose(s,l) picks a next machine state from a list of possible next machine states. The choice can depend upon the current machine state. (The matter of determining an optimal choose function is the subject of the next section.) The function $\Delta(m, s)$ returns machine state n with probability $\delta(m, s, n)$.

```
function execute($\mathcal{N}$,s)
  $m \leftarrow \mathcal{I}(s)$
  While type($m$) $\neq$ stop do
    if type($m$) $=$ action then
      $s \leftarrow$ environment($\pi(m)$)
    if type($m$) $=$ call then
      $s \leftarrow$ execute($\mathcal{N}, s$)
    if type($m$) $=$ choice then
      $m \leftarrow$ choose($m, s$))
    else
      $m \leftarrow \Delta(m, s)$
  return(s)
```

A further requirement for machines used by HAMs is that the call graph of each machine must be a tree. This means that there can be at most one path from any caller to any callee and implies that for any machine state, the call stack contents will be unique. This requirement prevents recursion in the HAM language, but it has the useful property of making a machine state a complete specification of the agent's execution status. If the call graph is a DAG, it can be converted to a tree by producing copies of the machines that can be called from more than one caller and then renaming the states in each copy.

A HAM is defined by an initial machine and the closure of all machine states in all machines reachable from the possible initial states of the initial machine. This chapter assumes that the top-level machine for a HAM does not have a stop state and, thus, never terminates. It also assumes that there are no infinite, probability 1 loops that never execute

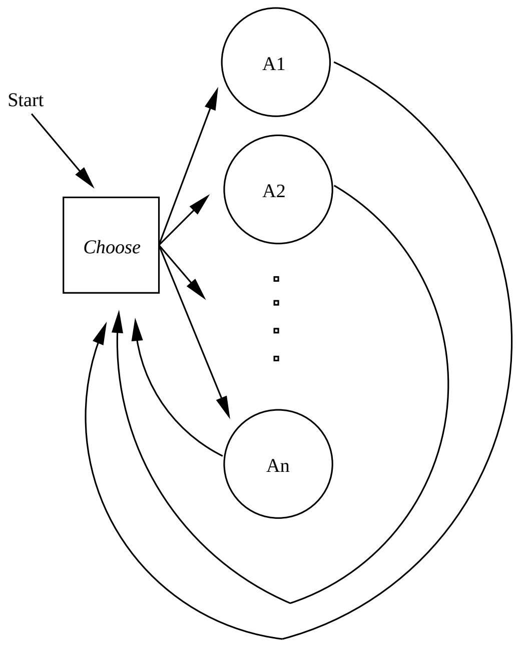

Figure 5.9: The null HAM.

an action. Such machines are syntactically valid according to the definitions described here and are possible even if the call graph is a tree. For example, machine $\mathcal{N}_1$ can call machine $\mathcal{N}_2$, which can move immediately to a stop state, return to control to $\mathcal{N}_1$ and proceed in an infinite loop. It is straightforward, but not particularly interesting, to generalize the results developed here to machines with top-level stop states or infinite, non-action loops.

As mentioned above, the HAMs can encode all types of abstract actions described in Chapter 3. They can encode the null abstract action that contains the same choices as the original MDP with a loop containing a single choice state that chooses between each possible action at every opportunity, as in Figure 5.9. HAMs can be designed to permit the concurrent use of abstract actions with low level actions, i.e., macros (Precup & Sutton, 1997) or options (Sutton et al., 1998), through choice states that select between actions and call states that execute temporally abstract actions.

5.6 Formal Properties of HAMs

This section presents the formal optimality and convergence properties of the HAM language when applied to an MDP.

Theorem 12 *For any MDP, $\mathcal{M}$, and any HAM, $\mathcal{H}$, there exists an SMDP, called $\mathcal{H} \circ \mathcal{M}$, the solution of which defines an optimal choice function,* choose(s,m)*, that maximizes the expected, discounted sum of rewards received by an agent executing $\mathcal{H}$ in $\mathcal{M}$.*

Proof: The proof proceeds by constructing an SMDP, $\mathcal{M}'$. In what follows, μ will refer to the closure of all states reachable from the initial machine for $\mathcal{H}$. The state space, $\mathcal{S}'$, of this SMDP will be pairs from $\mathcal{S}' = S \times \mu$. A state of $\mathcal{S}'$ is written $[s, m]$, where s is a state from the MDP and m is a machine state. A new transition function is constructed for $\mathcal{M}'$ using the transition function for $\mathcal{M}$ and the transition functions for $\mathcal{H}$. There are four cases for transitions from $[s, m] \in \mathcal{S}'$ to $[t, n] \in \mathcal{S}'$.

1. If m is an action state with $\pi(m) = a$, then there is only one possible action for s' and $\mathcal{T}'([s, m], [t, n]) = \mathcal{T}(s, t)\delta(m, t, n)$.

2. If m is a call state then $\mathcal{T}'([s, m], [s, n]) = \mathcal{I}_(n, s)$ for all n in $\pi(m)$, the machine called by state m.

3. If m is a choice state that selects between machine states $n_1 \ldots n_k$, then actions $c_1 \ldots c_k$ are defined for $[s, m]$ such that $\mathcal{T}'([s, m], c_i, [s, n_i]) = 1.0$ for all $1 \leq i \leq k$.

4. If m is a stop state, and l is the state on the call stack that invoked the machine containing m, then $\mathcal{T}'([s, m], [s, n]) = \delta(l, s, n)$.

All probabilities not specified above are 0. The reward function is defined as $R'([s, m], a) = R(s, a)$ when m is an action state and 0 otherwise. The discount factor is defined as $\beta'([s, m], a) = \beta(s, a)$ if m is an action state, and 0 otherwise.

The above construction satisfies the definition of an SMDP. The Markov property is preserved because of the tree-structured requirement on the call graph for $\mathcal{H}$. This ensures that there will always be a unique l on the call stack in item 4 above. The decisions in this SMDP correspond directly to choices of next machine states in the choose function for the machines in $\mathcal{H}$. Thus, the solution to this SMDP makes the optimal choices for every choice state. ■

Since $\mathcal{M}'$ is an SMDP with uncontrolled states, M' can be reduced to a smaller SMDP. A *choice point* is a machine state, environment state pair, $[s, m]$, such that m is a choice state.

Theorem 13 *For any MDP, $\mathcal{M}$, and any HAM, $\mathcal{H}$, there exists a reduced SMDP, called reduce($\mathcal{H} \circ \mathcal{M}$), that is equivalent to $\mathcal{H} \circ \mathcal{M}$, but contains no more states than the number choice points induced by $\mathcal{H}$ on $\mathcal{M}$.*

Proof: This follows from the previous theorem and from Theorem 7, which guarantees that all uncontrolled states can be removed from an SMDP to produce an equivalent SMDP. ∎

An important point that must not be lost in the details of this transformation is that the actions and the states in the transformed model still have meaning in the original model. State $[s, m]$ in $\mathcal{M}'$ corresponds to the situation where the agent is in machine state m and the environment is in state s. The action assigned to this state as the solution for $\mathcal{M}'$ corresponds to a decision made at choice state m. Another extremely important point is that the values assigned to the states of $\mathcal{M}'$ by the optimal policy for $\mathcal{M}'$ correspond directly to the expected, discounted sum of rewards received by an agent that takes the corresponding actions in the original model. This value may be less than the optimal solution to the original model, depending upon the quality of the machine hierarchy. In this sense, a HAM can be thought of as a means of constraining the set of policies that are considered for an MDP. The constraints are implicit in the structure of the machines in the HAM. As with any method that constrains the set of solutions for a problem, if the constraints are chosen poorly, the quality of the resulting solution will suffer.

These properties of HAMs should be considered in contrast to the properties of most state aggregation and function approximation methods. These methods solve a reduced problem that typically has, at best, an approximate relationship to the original MDP. The relationship between the solution to the reduced model and the solution to the original model is often quite loose. With HAMs, the value function for $reduce(H \circ M)$ indicates the true expected value of executing the policy in the *original* model.

The import of the theoretical results in this section deserves emphasis. This section has proved that the problem of optimally refining an incompletely specified, hierarchical, stochastic, finite state policy (a HAM) can be interpreted as an SMDP. Through SMDP state space reduction methods, the procedural knowledge contained in a HAM is used to produce a reduced problem. This provides the formal mechanism by which high-level strategies can be applied to new problems and efficiently refined to meet the needs of a new problem.

5.6.1 Model Reduction Mechanics

This section describes briefly how the theorems of the previous section are implemented efficiently. The greatest benefit will be reaped from the HAM approach when a HAM is combined with a model, $\mathcal{M}$, that has some distinguished start state $\mathcal{I}_{\mathcal{M}}$, at which all experiences in the environment begin. This reduces the number of reachable states in $\mathcal{S}'$. The reachable states in S' can be enumerated by performing a simple breadth-first search. For navigation problems, the all-at-once algorithm is the most efficient method of reducing the state space, since any state borders at most four other states, making the transition matrix nearly block diagonal for any policy.

The key to efficient use of the all-at-once algorithm is the identification of large, contiguous regions of $\mathcal{M}'$ that can be removed by solving a single system of equations. Two states, $[s, m]$ and $[t, n]$ belong in the same region if there is a path from $[s, m]$ to $[t, n]$ that does not pass through any choice states. Regions of this type can be identified efficiently in time that is linear in the number of edges in the state transition graph for $\mathcal{S}'$ with a simple search that tags states with tokens indicating to which region they belong. If a search from one state reaches a state that already is tagged with another region, the two regions are merged.

At a high level, the algorithm for constructing $reduce(H \circ M)$ is stated as follows, where aao(M,G) is a function that implements the all-at-once algorithm to remove region G from $\mathcal{M}$.

```
function reduce(H,M)
  $\mathcal{M}' \leftarrow H \circ M$
  $S' \leftarrow$ reachable states in $\mathcal{M}'$
  $G \leftarrow$ contiguous regions in $S'$
  For each $G_i$ in $G$
    $\mathcal{M}' \leftarrow$ aao$(M', G_i)$
  return($\mathcal{M}'$)
```

5.7 Reinforcement learning with HAMs

HAMs can be of even greater advantage in a reinforcement learning context, where the effort required to obtain a solution typically scales very badly with the size of the problem. The HAM constraints can focus exploration of the state space, reducing the "blind search" phase that reinforcement learning agents must endure while learning about a new environment. Learning also will be faster since the agent effectively is operating in a reduced state space.

This section introduces a variation of Q-learning called HAMQ-learning that learns directly in the reduced state space *without performing the model transformation* described in the previous section. This is significant because the environment model is not usually known *a priori* in reinforcement learning contexts.

A HAMQ-learning agent keeps track of the following quantities: t, the current environment state; n, the current machine state; s_c and m_c, the environment state and machine state at the previous choice point; c, the choice made at the previous choice point; and r_c and β_c, the total accumulated reward and discount since the previous choice point. It also maintains an extended Q-table, $Q([s,m],c)$, which is indexed by an environment-state/machine-state pair and by an action taken at a choice point.

For every environment transition from state s to state t with observed reward r and discount β, the HAMQ-learning agent updates: $r_c \leftarrow r_c + \beta_c r$ and $\beta_c \leftarrow \beta\beta_c$. For each transition to a choice point, the agent does

$$Q([s_c,m_c],c) \leftarrow Q([s_c,m_c],c) + \alpha_{[s_c,m_c]}[r_c + \beta_c V([t,n]) - Q([s_c,m_c],c)],$$

and then $r_c \leftarrow 0$, $\beta_c \leftarrow 1$.

Theorem 14 *HAMQ-learning will converge to the optimal policy for reduce*$(H \circ M)$ *with probability 1 when* $\sum_i \alpha_i(s) = \infty$ *and* $\sum_i \alpha_i^2(s) < \infty$.

Proof: This follows from the fact that *reduce*$(H \circ M)$ is an SMDP and that Q-learning converges for SMDPs (Theorem 2).■

5.8 Experimental results

This section describes some experimental results using the HAM approach on a large navigation MDP with just under 3700 states, as shown in Figure 5.10. The domain is full of obstacles of the type shown in Figure 5.11. There are four possible actions in this domain, corresponding to the four coordinate directions. The transition model specifies that each action succeeds 80% of time, while 20% of the time the agent moves in an unintended perpendicular direction. The agent begins in a start state in the upper left corner. A reward of 5.0 is given for reaching the goal state and the discount factor β is 0.999.

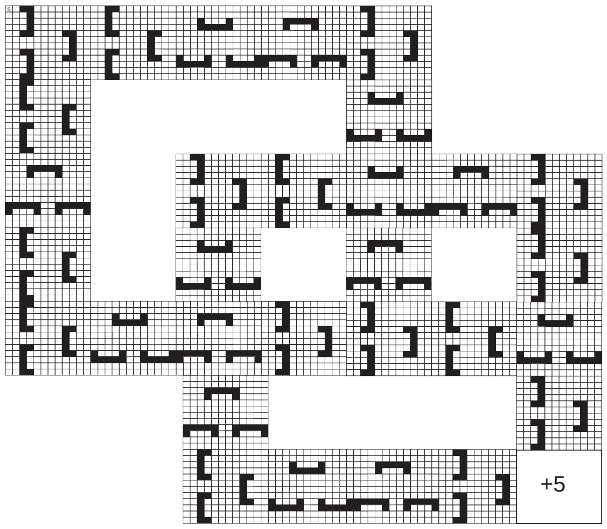

Figure 5.10: A navigation MDP with ≈ 3700 states.

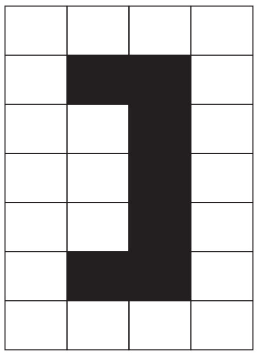

Figure 5.11: An obstacle.

The HAM for the experimental domain contained four machines for navigating the hallways in each of the four directions. Each machine moved the agent in the desired direction until the end of the hallway or an intersection is reached. When an obstacle is encountered, a *choice point* is created to choose between two possible next machine states. One calls a backoff machine to back away from the obstacle and then move forward until the next one. The other calls a follow-wall machine to try to get around the obstacle. The follow-wall machine is very simple and will be tricked by obstacles that are concave in the direction of intended movement; the backoff machine, on the other hand, can move around any obstacle in this world but could waste time backing away from some obstacles unnecessarily and should be used sparingly.

The complete "navigation HAM" involves a three-level hierarchy, somewhat reminiscent of a Brooks-style architecture but with hard-wired decisions replaced by choice states. The top level of the hierarchy is basically just a choice state for choosing a hallway navigation direction from the four coordinate directions. This machine has control initially and regains control at intersections or corners. The second level of the hierarchy contains four machines for moving along hallways, one for each direction. Each machine at this level has a choice state with four basic strategies for handling obstacles. Two back away from obstacles and two attempt to follow walls to get around obstacles. The third level of the hierarchy implements these strategies using the primitive actions.

The transition function for this HAM assumes that an agent executing the HAM has access to a short-range, low-directed sonar that detects obstacles in any of the four axis-parallel adjacent squares and a long-range, high-directed sonar that detects larger objects such as the intersections and the ends of hallways. The HAM uses these partial state descriptions to identify feasible choices. For example, the machine to traverse a hallway up would not be called from the start state because the high-directed sonar would detect a wall in the up direction. The sonars help reduce the number of obstacle avoidance choices in most situations from 4 to 2.

The navigation HAM represents an abstract plan to move about the environment by repeatedly selecting a direction and pursuing this direction until an intersection is reached. Each machine for navigating in the chosen direction represents an abstract plan for moving in a particular direction while avoiding obstacles. The optimal refinement of this abstract strategy picks the best action for every place where the HAM induces a choice point.

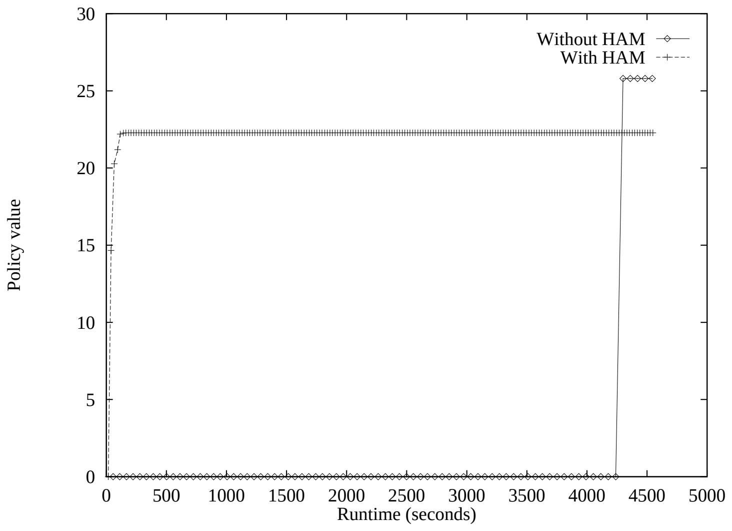

Figure 5.12: Ordinary policy iteration, and policy iteration with HAMs

Figure 5.12 Shows the run time in seconds for policy iteration to solve the original problem and for policy iteration to solve the reduced problem. The vertical axis shows the quality of the policies as measured in expected discounted sum of rewards. The graph does not show the time required to construct the reduced model. However even when this 866 second cost (which was obtained in fairly sloppy lisp code) is added in, the HAM method produces a good policy in less than a quarter of the time required to find the optimal policy in the original model. The actual solution time is 185 seconds versus 4544 seconds.

Experiments with HAMQ-learning were even more dramatic. In these tests, exploration was achieved by selecting actions according to the Boltzmann distribution with a temperature parameter for each state. An inverse decay was used for the learning rate, α. Figure 5.13 compares the learning curves for Q-learning and HAMQ-learning. HAMQ-learning appears to learn much faster: Q-learning required 9,000,000 iterations to reach the level achieved by HAMQ-learning after 270,000 iterations. Even after 20,000,000 iterations, Q-learning did not do as well as HAMQ-learning. The poor performance of Q-learning appears to be related, at least in part, to the problem of blind search. Once HAMQ-learning starts moving down a hallway, it keeps trying to move forward until it reaches the end of the Hallway. This forces HAMQ-learning to cover large expanses of the state space early in the learning process. In contrast, Q-learning can waste large amounts of time roaming aimlessly in the top-left corner of the domain before enough randomly chosen actions move it

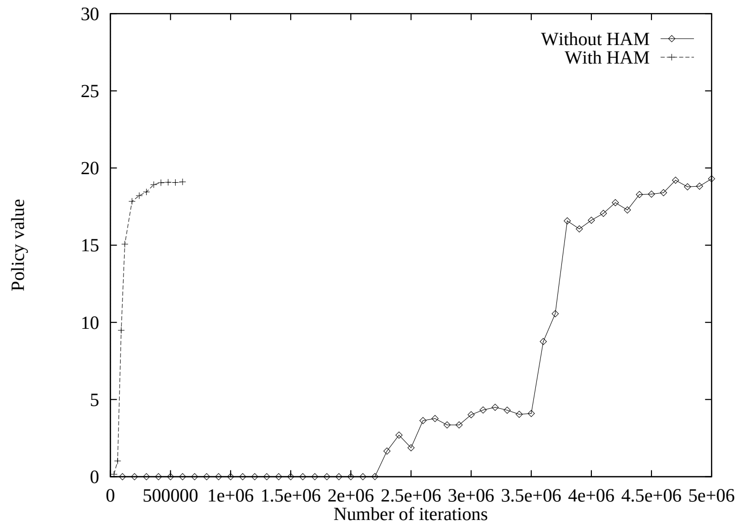

Figure 5.13: Value the learned policy at the initial state for HAMQ-learning and regular Q-learning.

towards the goal area. Q-learning also must learn more parameters. Since HAMQ-learning was operating in a reduced state space, it learned only 4,300 parameters, the product of the number of choice points and the number of choices. Ordinary Q-learning was forced to learn nearly 14,000 parameters, 4 for each state.

The best feature about this example of HAMs is that the navigation HAM achieved an impressive performance boost with a minimal amount of domain knowledge. The same HAM could be used in any navigation-type domain with a similar sensor model. HAMs are "plug and play."

5.9 Bootstrapping

This chapter established that a HAM $\mathcal{H}$ can be combined with an MDP $\mathcal{M}$ to produce a new MDP, $\mathcal{M}' = \mathcal{H} \circ \mathcal{M}$ and that the optimal policy $\pi^{*\prime}$ for $\mathcal{M}'$ corresponds to a non-stationary policy in $\mathcal{M}$. Of course, there is no guarantee that $\pi^{*\prime}$ will be optimal for $\mathcal{M}$. The solution to the reduced MDP will find the optimal policy that is consistent with the HAM constraints, but this policy will be only as good as allowed by the constraints implicit in the HAM.

What if the best policy consistent with the $\mathcal{H}$ is not good enough? One option is to loosen some of the constraints imposed by the HAM by introducing more choice points.

More sophisticated machines incorporating more prior knowledge about the domain could be introduced as well. When all else fails, however, it may be necessary to revert to the full, original MDP and under such circumstances one would like to recover some of the effort invested in constructing the HAM policies.

Examination of an iterative method like policy iteration shows where the information gained by solving a HAM policy can be used to "bootstrap" the solution to the original MDP. Recall the performance graph for policy iteration from in Figure 5.12. There are two important characteristics:

1. Once a good (non-zero) policy is found, the optimal policy is found shortly thereafter.
2. The optimal HAM policy is found before any good policies are found for the original MDP

This suggests that constructing a good stationary policy from a non-stationary HAM policy, then using this policy to bootstrap policy iteration (or some other iterative method) may provide a faster path to the optimal policy for the original MDP than solving the original MDP directly. Recall that as part of the all-at-once algorithm, a value function is computed that expressed the value of every removed state in terms of the neighboring choice points. If these expressions are saved, they can be used to construct a value function for the original, unreduced space, from the solution to the reduced problem. Call this value function $V^{'*}$.

Theorem 15 *For any MDP, $\mathcal{M}$, and HAM, $\mathcal{H}$, let $\pi^{*\prime}$ be the optimal policy for $\mathcal{M}' = \mathcal{H} \circ \mathcal{M}$, let $V^{*\prime}$ be the corresponding value function, and let V^* be the optimal value function for $\mathcal{M}$, then for any s, $V^*(s) \geq \max_m V^{*\prime}([s,m])$ and $\max_m V^{*\prime}([s,m])$ can be used to construct a stationary policy with value at least $\max_m V^{*\prime}([s,m])$.*

Proof: Consider the following modification to the HAM execution routines: Whenever the agent is at a non-choice state h, and environment state s, it can execute the action specified by h, or it can search the value function, $V^{*'}([s,m])$ for all extended states containing s. If there exists an n such that $V^{*'}([s,m]) > V^{*'}([s,n])$, then the agent "jumps" to machine state n and changes the call stack to appear as if the agent had arrived upon machine state n directly. This process defines a stationary policy π for $\mathcal{M}$ since it assigns an action to every state that is independent of the agent's machine state. Moreover, $V_\pi(s) \geq \max_m V^{*'}([s,m])$ for all s since the agent will do at least as well as $\max_m V^{*'}([m,s])$ by switching to state m.
■

HAM policies can also be used to bootstrap on-line MDP methods. The most straightforward approach would be the use of an off-line HAM solution to initialize a value function for an on-line learner, similar to the manner in which it could be used to initialize a value function for value iteration. The applicability of this approach is limited since it assumes that the model is available to the off-line algorithm, while in many reinforcement learning scenarios an explicit model is presumed to be unavailable.

If the model is not known, then a learning agent must construct some kind of a model or value function for the non-choice states if values for these states are to be used to bootstrap a solution to the full MDP. There are several possibilities:

1. Learn in the extended (but not reduced) state space.
2. Simultaneously learn separate value functions for both the extended and reduced state space.
3. Learn a domain model while using another method, like Q-learning, to learn the optimal policy for the extended space; then use the model to compute $V^{*'}([s,m])$ offline.
4. Use symbolic TD to learn an algebraic expression for the values of the non-choice states.

The first option is the most straightforward, but will likely suffer from the problem of slow convergence due to the potentially large size of the extended space. The second option, which would, in parallel, combine the HAMQ-learning algorithm with standard temporal difference learning in the extended state space, has the potential for faster convergence. In this approach, Q-learning would be used to learn the values of the choice points in the reduced space and then temporal difference learning would be used to propagate these values back to the non-choice points in the full, extended state space. This is illustrated in Figure 5.14. Note that there is no flow of information from the non-choice states to the choice states. One can think of a separate TD process eavesdropping on the HAMQ-learning process.

The third option would require the least additional work from the agent while the initial HAM policy is learned, but would require substantial additional effort to convert the HAM policy to a policy or value function for the original MDP. The agent would be forced

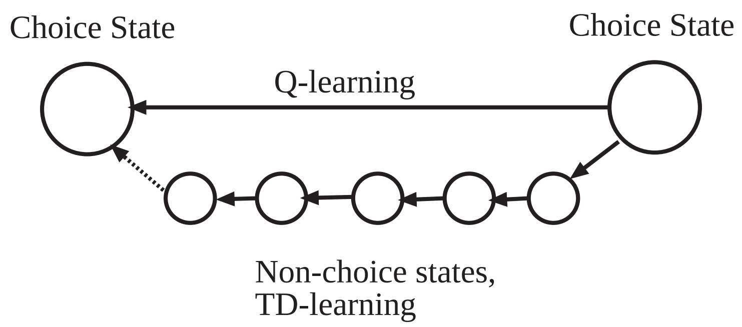

Figure 5.14: Mixed TD and Q-learning. The dashed line indicates that no reinforcement is performed from non-choice states to choice states. Choice state reinforcement is handled by the Q component.

to construct the extended state space off-line, an awkward approach if the agent is situated and is expected to respond in real time.

The fourth option involves the use of symbolic TD. In this application, there is no need to freeze the policy since the states in question are non-choice states. An important difference between the symbolic approach and the hybrid Q/TD approach (option 2) is that the learning of the optimal policy for the reduced space is completely decoupled from the learning of an algebraic expression for the value of the non-choice states. The two learning processes execute in parallel with no communication between them. Symbolic TD may converge faster than traditional TD in this context because meaningful information about the relationship between non-choice states and choice-states can be propagated back well before the agent has discovered a good policy for the choice states.

Finally, another interesting issue is raised by the availability of a policy that may be superior to the optimal HAM policy: If the intermediate state values needed to construct efficiently the value function of Theorem 15 are available, should the stationary policy be constructed and used in place of the HAM policy? The answer is an emphatic *yes*, since the theorem guarantees that it can only improve performance. However, the application of this theorem in practice will depend upon the memory requirements for storing explicit information about each of the extended states.

5.10 Conclusion

This section presented Hierarchies of Abstract Machines (HAMs). HAMs incorporate prior knowledge with temporal abstraction methods to produce an abstract or partially specified plan for an action. HAM action descriptions can be optimally refined by creating an SMDP, the solution of which replaces every point at which the HAM is incompletely specified with a concrete decision. HAMs can contain information that reduces choices that are available to the agent in many situations. This information is used to construct a reduced state space, making the problem of refining a HAM for a particular task much simpler than devising a new policy from scratch. Since the task of refining a HAM is interpreted as an SMDP, provably optimal and convergent on-line and off-line algorithms can be used to optimally refine a HAM. HAM solutions also can be used to bootstrap traditional MDP algorithms.

Chapter 6

Problem Decomposition

The previous chapter presented the HAM method for optimally refining hierarchical, partially specified, temporally abstract actions for MDPs. While HAMs induce a hierarchical partitioning of the state space and impose a hierarchical ordering on the choices made while executing the HAM, they do not lead directly to a hierarchical solution method for the resulting SMDP, and they do not decompose the original MDP into independent pieces.

This chapter addresses the topic of decomposing MDPs into independent subproblems. It provides two new approaches to decomposing and solving MDPs as a collection of separate subproblems. The first of these methods builds a cache of policies for each part of the problem independently, and then combines the pieces in a separate, light-weight step. The second method also divides the problem into smaller pieces, but information is communicated between the different problem pieces, allowing intelligent decisions to be made about which piece requires the most attention. Both approaches can be used to find optimal policies or approximately optimal policies with provable bounds. These algorithms also provide a framework for the efficient transfer of knowledge across problems that share similar structure.

The algorithms in this chapter are complementary to the HAM approach of Chapter 5. HAMs induce a partitioning of the MDP state space based upon the active machine subroutine. The algorithms and theorems of this chapter show how each HAM subroutine call can be treated as an independent subproblem.

6.1 An Introduction to Decomposition

In the hallway navigation example of the previous chapter, the problem of moving right across the top hallway of the model is not isolated completely from the rest of the MDP. The value of any choice made at any point along the hallway can be influenced by the values of the states at the end of the hallway. These are in turn influenced by the solution to the adjacent vertical hallway segment, which is, itself, connected to the rest of the problem.

HAMs *organize* the solution process by introducing a hierarchical set of choices, and they *reduce* the state space by introducing knowledge and skipping over states that do not contain choices. However, since HAMs transform one (S)MDP to another SMDP, they do not necessarily remove the dependencies that can exist between different states in the model. If there were a way of removing these dependencies, it would be possible to view each HAM subroutine call as a separate subproblem that could be solved once, in isolation from the rest of the problem, and never revisited. Since the run time for MDP algorithms grows very rapidly in the size of the state space, this decomposition could lead to huge computational benefits by replacing large MDPs with a number of smaller ones, averting the need to ever solve a large MDP directly in its entirety.

There is reason to believe that the type of decomposition outlined above is not possible *in general.* This type of decomposition would make MDPs highly parallelizable, permitting different chunks of the state space to be solved as independent subproblems on different processors, and admitting a linear speedup in the number of processors assigned to the problem. Papadimitriou and Tsitsiklis (1987) show that parallelizing MDPs is as hard computationally as parallelizing linear programming, which is in a subclass of problems in P that is believed to be the least amenable to parallelization.

While some progress has been made in understanding some very special cases where MDPs may be decomposed into independent subproblems (Singh, 1992; Lin, 1997), much of the effort has focused on methods that decompose MDPs into "communicating" subproblems (Bertsekas & Tsitsiklis, 1989; Dean & Lin, 1995) (Chapter 2 and Chapter 3). In these iterative methods, information about subproblem solutions is communicated to neighboring subproblems. However, the solution for each subproblem may need to be updated many times until a globally optimal solution is obtained.

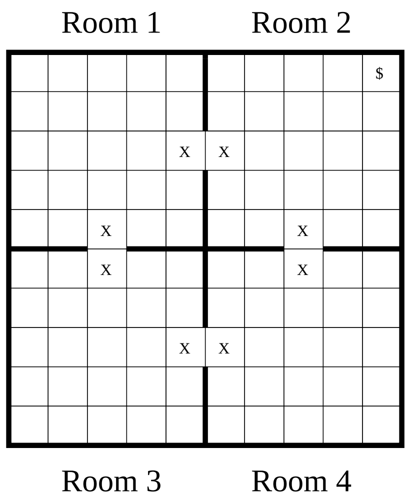

Figure 6.1: A weakly coupled MDP. There is a reward in room 2, indicated with a \$. Connecting states are identified with an X.

This chapter considers a special, but fairly general, class of problem decompositions where each subproblem is "weakly" coupled with the neighboring subproblems. This means that the number of states connecting the two subproblems is small, a relationship that appears naturally in many problems. For example, the problem of moving from one's office to one's house has this structure: one's office is a small region that is connected by a much smaller region, the door, to an external corridor. Many other offices may be connected to this corridor, each with a similar structure. The corridor could be fairly large and connected to other corridors by relatively small intersection regions. Most buildings have a small number of doorways that connect them to the streets outside. Each street has a relatively small number of points where it connects to other streets. One such street connects to the house one calls home, which is itself an aggregation of weakly connected pieces. An MDP is weakly coupled if it can be divided into two or more subproblems that are weakly coupled with each other.

Figure 6.1 shows the familiar navigation MDP divided into four rooms, each of which can be considered a subproblem.

This chapter uses a similar approach to that used in communicating MDP solution methods, but aims to avoid iteratively updating solutions to subproblems by building a set of policies independently for each subproblem. Each set of policies is called a *cache*. The caches are constructed in such a way that they are guaranteed *a priori* to provide performance within a constant of the optimal, regardless of the structure of the other subproblems. This permits a complete decoupling of the MDP into independent subproblems that can be solved in parallel and then recombined in a light-weight step. The decoupling process is based upon the observation (Chapter 3) that any policy over a region of state space defines a linear function for the values of the states inside the region in terms of the values of the states outside the region. The linear relationship is exploited by the algorithms in this chapter to build caches for each region of the state space. The caches are built iteratively by constructing linear programs that discover the values of the states outside the region for which cache performs the worst, then adding a new policy to the cache to cover the worst case.

The efficient manipulation of policy caches also provides a formal basis for the transfer of knowledge across problems with similar substructures. The simplest case of this occurs when the reward structure for a problem changes. Suppose, for example, that the reward in the navigation problem is moved from room 2 to room 4. Policy caches devised for rooms 1 and 3 can be transferred to the new problem. Similarly, if one's destination is now a cafe instead of home, the policies designed for one's office and the containing building should transfer to the new problem.

Since the number of possible policies for a subproblem is exponential in the number of states in the subproblem, there may exist problems and accuracy requirements for which the size of the policy cache will be exponential. In these cases there still will be some benefit to constructing a small policy cache, even if it does not provide the desired accuracy guarantees. This chapter presents an algorithm that augments standard communicating MDP algorithms with the use of a policy cache. The policy cache can be used to determine lower *and* upper bounds on the values that states in the subproblem can assume, and this provides a means of deciding when it is worth using a cached solution and when it is worth producing a new subproblem solution. This is particularly useful in determining if subproblem solutions from a related problem can be applied to a new one.

These concepts are introduced first in the context of a normal MDP, but are extended in the end of the chapter to exploit the hierarchical information contained in HAMs.

6.2 Review of Decomposition Algorithms

The first step for a decomposition method for MDPs is the division of the state space into disjoint subsets, $G_1 \ldots G_m$. In the simple navigation problem of Figure 6.1, each room could be a subset. For each subset, G_i, there exists a set of states not in G_i that are reachable in one step from G_i. Call this the *out-space* of G_i. In Figure 6.1, the out-space of the top-left room contains one state in the room to the right, and one in the room below. The *in-space* of region G is defined as the set of states inside of G reachable in one step from a region outside of G.

The next step is the introduction of a set of policy caches, $\Pi_1 \ldots \Pi_m$, defined over each of the regions. Each cache contains a set of policies that can be used in the corresponding reach of the state space. Using the SMDP techniques of Chapter 3 the optimal assignment of policies to regions can be determined by solving a "high-level" reduced SMDP defined over only the states in the out-spaces of the regions. The reduced problem removes all but the out-space states from the problem. Actions in the reduced problem correspond to assignments of policies to regions in the original decision problem.

In Figure 6.1, the high-level SMDP would contain just the eight specially marked states. An action in the high level problem would correspond to a decision to adopt some policy from the cache upon entering a room, and to stay with this policy until the next out-space state is reached. The solution to the high-level problem may produce a *non-stationary* policy at the low-level, which means that the actions taken in any room may depend upon the manner in which the room is entered. A non-stationary policy of this type can be converted easily to a stationary policy that is at least as good (Chapter 5).

The relationship between the size of the out-spaces and the complexity of the high-level problem should make the importance of weak coupling clear. If the size of the out-spaces approaches the size of the original MDP, then the high-level SMDP will be as difficult as the original MDP.

An algorithm that *completely* decomposed an MDP would produce a Π_i for each G_i, combine these to produce an optimal or approximately optimal overall solution, and never need to revise any of the Π_i. Unless the Π_i are chosen very carefully, or the caches are very large, combinations of policies in the initial policy caches may not suffice. There are several approaches to revising the policy caches. The Macros/Options literature (Sutton et al., 1998) takes one extreme end of this spectrum, where policies and low-level actions are mixed together in the same SMDP. This sacrifices the reduction in computational complexity obtained from solving a reduced decision problem in favor of a guarantee of obtaining optimality. The iterative abstraction approach in Dean and Lin (1995) updates each Π_i directly. Dean and Lin considered a special case in which a new policy was computed for each region at each iteration based upon the high-level decision problem's current estimates for the value of the out-space states. (See Chapter 3.) Note that the high-level decision problem was actually a trivial value determination problem and not really a decision problem since $|\Pi_i| = 1$ for all i.

The algorithms in this chapter all aim to minimize the number of policies that are computed for MDP subproblems. The extent to which this can be minimized is a measure of how effectively an MDP has been decomposed. If each subproblem requires only a small cache of candidate solutions, this means that the subproblem solutions are relatively independent. These are precisely the situations in which a large computational benefit is reaped from decomposition, since the MDP can be divided and conquered by solving a reasonable number of small subproblems. The size of the policy caches also gives some measure of the parallelizability of the problem. If a region can be solved with a small cache of policies, this suggests that the entire cache could be constructed *a priori* as a completely independent subprocess.

The following section describes several algorithms for constructing policy caches with minimal knowledge of how the subproblem is connected to overall MDP. These algorithms aim to minimize the size of the cache, while ensuring that solutions using the cache will be within a bound of optimal. The succeeding section describes a scheme for working with policy caches for which optimality bounds have not been established *a priori.* This method efficiently establishes bounds on the benefit of adding a new policy to a cache, based upon the current contents of the cache.

6.3 Complete decoupling

This section presents algorithms that find a policy cache, Π, for a particular region, G, such that Π is guaranteed to provide policies that are within a constant of optimal when a high level SMDP using Π for G is solved. The only assumptions that are made about the regions to which G connects is that the states have values on $[V_{\min} \ldots V_{\max}]$.

Define $\mathcal{V}^{\mathcal{O}}_{G}$, as a vector of values that the states in the out-space of G can take on (the subscript will be dropped when there can be no confusion about the region in question). The *fan-out* of a region is defined as the dimension of this vector, which will be denoted by the variable d.

In addition to storing a cache of policies it is useful to store a cache of functions, $f_{\pi_i}(s, \mathcal{V}^{\mathcal{O}})$ for each $\pi_i \in \Pi$. Each $f_{\pi_i}(s, \mathcal{V}^{\mathcal{O}})$ is a linear function that provides the value of any state $s \in G$ as a linear function of $\mathcal{V}^{O}$. For any policy functions can be determined by using any of the algorithms in Chapter 4.

The goal in constructing a policy cache for a region is to produce a cache such that for every possible value of the corresponding out-space states, there is a policy in the region's cache for which the performance in the region will be within a bound of optimal. A policy, π, for region G is optimal with respect to $\mathcal{V}^{O}$ if π is the solution to the MDP defined just over the states in G, with the assumption that states in the out-space of G are absorbing states with values locked at the value of the corresponding entry in $\mathcal{V}^{O}$. In room 1 of the four-room example, the optimal policy for $\mathcal{V}^{O}$ would be determined by solving an MDP with just the states in room 1 and the two connecting states in room 2 and room 3. The value of the connecting state in room 2 would be treated as a constant with value $\mathcal{V}^{\mathcal{O}}[0]$ and the value of the connecting state in room 3 would be a constant with value $\mathcal{V}^{\mathcal{O}}[1]$. A policy, π, is said to be ϵ-optimal with respect to $\mathcal{V}^{\mathcal{O}}$ if $\forall s \in G$, $BE(V_\pi) \leq \epsilon$ when the values of the states in the out-space of G are fixed by $\mathcal{V}^{O}$.

For any state and any value of $\mathcal{V}^{\mathcal{O}}$, there must be one policy in the cache that appears at least as good as all of the others. A policy, π, *dominates at t* for a particular $\mathcal{V}^{\mathcal{O}}$, if $f_\pi(t, \mathcal{V}^{\mathcal{O}}) \geq f_{\pi_j}(t, \mathcal{V}^{\mathcal{O}})\ \forall j$. This means that the low-level policy, π, appears to be the best high level action at state t for a particular $\mathcal{V}^{O}$. A cache of policies is *$\epsilon-$optimal at t* if, for any $\mathcal{V}^{\mathcal{O}}$, the dominating policy is $\epsilon-$optimal. A cache of policies is $\epsilon-$optimal if it is $\epsilon-$optimal at all t in the in-space of G.

Theorem 16 *If an MDP is divided into m regions, $G_1 \ldots G_m$ and an $\epsilon-$optimal cache of policies, $\Pi_1 \ldots \Pi_m$, is determined for each region, these policies can be combined to produce a globally $\frac{\epsilon}{(1-\beta)}-$optimal policy by solving an SMDP with at most $\sum_i |\mathcal{V}^{\mathcal{O}}_{G_i}|$ states and $\sum_i |\Pi_i|$ actions.*

Proof: As in Section 6.2 and Chapter 3, a high-level SMDP can be defined over the states $U = \bigcup_i G_i$. The solution to this decision problem assigns values to the states in U and assigns dominating policies to each state in the in-space of each region in U based upon these values. Consider an arbitrary region G and the policy, π, assigned to it when it is entered at state t. By the definition of $\epsilon-$optimality, $BE(V_\pi(s)) \leq \epsilon$ for all s in G. Since this will be true for all states in all regions, the maximum Bellman error for any policy that will be used in any region will be less than ϵ, which means that the maximum Bellman error for the entire problem will be less than ϵ, which is sufficient to ensure $\frac{\epsilon}{(1-\beta)}-$optimality, using the error bounds from Chapter 2.■

Note that the policy produced in the above theorem is non-stationary. The policy used in a region can depend upon how the region is entered. For example, the policy used in room 1 when room 1 is entered from the bottom may be different from the policy used when room 1 is entered from the right. This is not a problem since the above theorem shows that no matter how the room is entered, the value function corresponding to the adopted policy will have a Bellman error of no more than ϵ.

This theorem provides a means of combining a set of approximately optimal solutions to produce a global solution that is also approximately optimal. Of course, if $\epsilon = 0$, then the solution will be optimal. The following subsections will describe three algorithms for constructing $\epsilon-$optimal policy caches.

6.3.1 The $\epsilon-$grid Approach

The most straightforward approach to devising a policy cache, as described in Hauskrecht et al. (1998), is to create an ϵ resolution grid over the space of possible values for $\mathcal{V}^{\mathcal{O}}$, then produce an optimal policy with respect to each grid point. This is sufficient because the value of any policy can change by at most ϵ when moving within one cell of the grid in $\mathcal{V}^{\mathcal{O}}$ space. This bounds the loss from using a policy defined for the nearest grid point instead of the optimal policy. This result also is established formally using an alternative argument in Hauskrecht (1998).

Lemma 2 *The sum of the coefficients of any f_π must be less than* 1.

Proof: Consider the effects of policy π when applied at state s of region G. Call the states in the out-space of G, $s_1 \ldots s_d$. π will reach state s_i with probability p_i. The coefficient of f_π that determines $V_\pi(s)$ in terms of the value of $V(s_i)$ can be no larger than p_i. Since $\sum_i p_i \leq 1$, the sum of the coefficients of f_π can be no more than 1. ■

This bound on the coefficients of f_π limits how quickly the value of any state can grow as the values of the states in the out-space grow.

Theorem 17 *Consider two grid points $\mathcal{V}_1$ in $\mathcal{V}_2$ in the same cell of an ϵ-grid covering of $\mathcal{V}^{\mathcal{O}}$. Let $\mathcal{V}_1$ be the lowest corner of the grid cell, and let $\mathcal{V}_2$ be any other point in the same cell. Let π_1 and π_2 be optimal with respect to $\mathcal{V}_1$ and $\mathcal{V}_2$ respectively, then $f_{\pi_2}(s, \mathcal{V}_2) \leq f_{\pi_1}(s, \mathcal{V}_1) + \epsilon$ for all s.*

Proof: First, note that if $\pi_1 = \pi_2$, the lemma holds. Since the coefficients of any f_π must sum to less than 1, the value of any state under the same policy can grow by at most ϵ when moving away from $\mathcal{V}_1$ within the same cell, since each element of $\mathcal{V}^{\mathcal{O}}$ can change by at most ϵ. Suppose $\pi_1 \neq \pi_2$ and $f_{\pi_2}(s, \mathcal{V}_2) > f_{\pi_1}(s, \mathcal{V}_1) + \epsilon$ for some s. Since f_{π_2} can grow by at most ϵ in moving from $\mathcal{V}_1$ to $\mathcal{V}_2$, this implies $f_{\pi_2}(s, \mathcal{V}_1) > f_{\pi_1}(s, \mathcal{V}_1)$. However, f_{π_1} is optimal with respect to $\mathcal{V}_1$, so this is a contradiction.■

The main problem with the $\epsilon-$grid approach is that it can require a huge number of policies, $(\frac{V_{\max} - V_{\min}}{\epsilon})^d$. This will be unmanageable unless the range of values is very small, the fan-out of the region is very small, or ϵ is very large.

6.3.2 Value Space Search

This section presents an algorithm that aims to avoid constructing an exponential number of policies by searching through $\mathcal{V}^{\mathcal{O}}$ space to find a point at which the current policy cache is not adequate. If such a point is found, a new policy is added to the cache, and the process is repeated until no points can be found for which the current cache is inadequate. The following formal results are the basis of the value space search algorithm:

Lemma 3 *For any state s, the dominating policies at s form a piecewise-linear convex function of $\mathcal{V}^{\mathcal{O}}$.*

Proof: This follows from the observation that using the best policy means taking the maximum over a set of linear policy functions. ■

This lemma is demonstrated with a simple example using the model in Figure 6.2, where there is a room with just one exit, but a +1 absorbing reward state in the center of the room. Possible actions are right, left, up and down, but these actions are unreliable, resulting in movement in one of the three other axis-parallel directions 20% of the time.

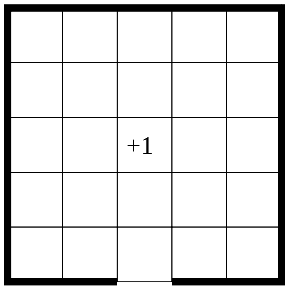

Figure 6.2: A simple MDP with a one state out-space.

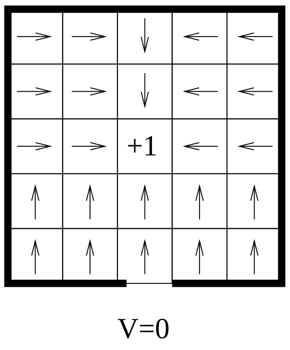

Figure 6.3: The optimal policy when the out-state has value 0.

The discount factor used was 0.95. Note that there are more than two interesting policies for this subproblem. If the value of the out state is very large, the optimal policy will move along the perimeter of the room to avoid accidentally hitting the reward in the center. As the value of the out-state gets closer to 1, the optimal policy becomes less conservative and will risk brushes with the +1 reward state. Figure 6.3 shows the optimal policy for $\mathcal{V}^{\mathcal{O}} = [0]$. This policy moves towards the absorbing +1 reward. Figure 6.4 shows the optimal policy when the value for $\mathcal{V}^{\mathcal{O}} = [1.09]$. This policy prefers to exit the room, but will sometimes go for the +1 absorbing state if it is closer, due to the discount placed on future rewards. Figure 6.5 shows the optimal policy for $\mathcal{V}^{\mathcal{O}} = [2]$. This policy tries to avoid the absorbing state completely.

The value function for this room can be displayed in two dimensions since $\mathcal{V}^{\mathcal{O}}$ is just a scalar. Figures 6.6 through 6.8 show the value surface as policies are added to the

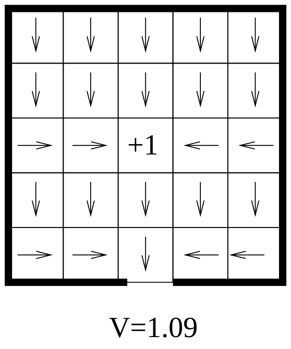

Figure 6.4: The optimal policy when the out-state has value 1.09.

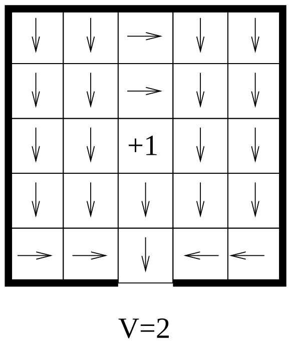

Figure 6.5: The optimal policy when the out-state has value 2.

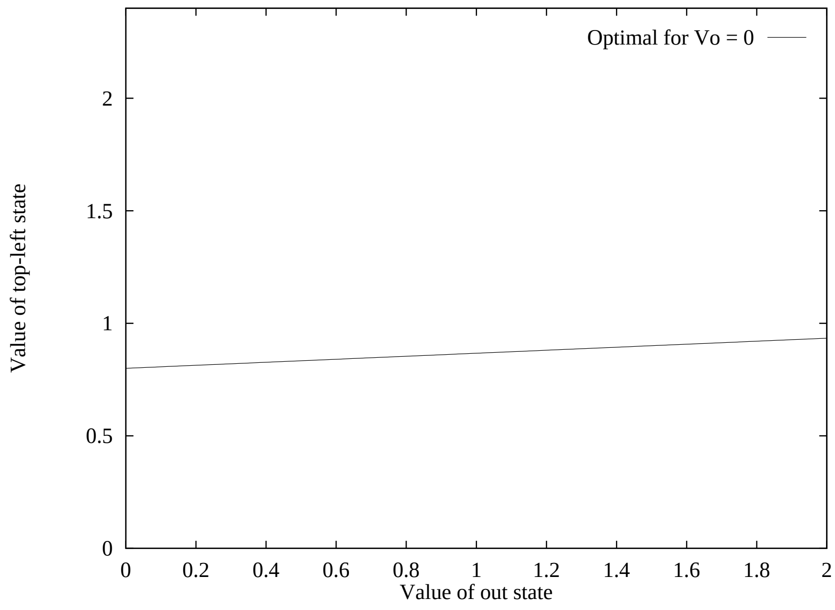

Figure 6.6: The optimal policy for $\mathcal{V}^{\mathcal{O}} = [0]$ is added to the cache. This policy avoids the exit, making the value of the top-left state nearly independent of the value of the exit.

cache for this room.

The following result establishes a means by which the quality of a cache of policies can be checked efficiently.

Theorem 18 *For region G, with n states, a actions, and policy cache, $\Pi = \pi_1 \ldots \pi_m$, the point in $\mathcal{V}^{\mathcal{O}}$ space for which the Bellman error of the dominating policy is largest, can be determined in time that is polynomial in n, a, m, and d.*

Proof: This is achieved by means of a linear program. For all t in the in-space of G, for all s in G, for all a, and for all $\pi \in \Pi$, the following linear program is solved:

Maximize:

$$\mathcal{R}(s,a) + \sum_{s'} \mathcal{T}(s,a,s') f_\pi(s', \mathcal{V}^{\mathcal{O}}) - f_\pi(s, \mathcal{V}^{\mathcal{O}})$$

Subject to:

$$\begin{aligned} f_{\pi_i}(t, \mathcal{V}^{\mathcal{O}}) &\le f_\pi(t, \mathcal{V}^{\mathcal{O}}) \ \forall \pi_i \in \Pi \\ \mathcal{V}^{\mathcal{O}}[i] &\le V_{\max},\ 1 \le i \le d \end{aligned}$$

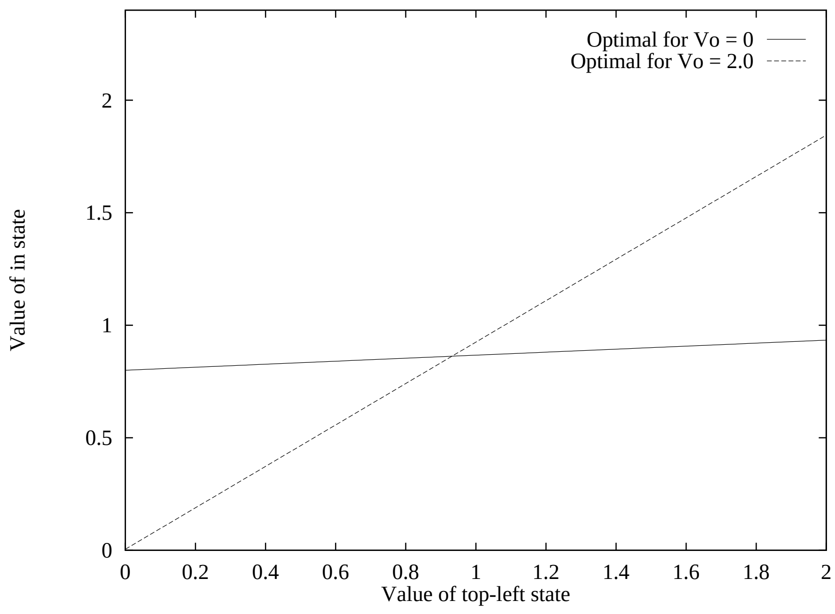

Figure 6.7: A second policy that is optimal at $\mathcal{V}^{\mathcal{O}} = [2]$ is added to the cache. This policy has a strong dependence on $\mathcal{V}^{\mathcal{O}}$ since it sends the agent directly to the exit.

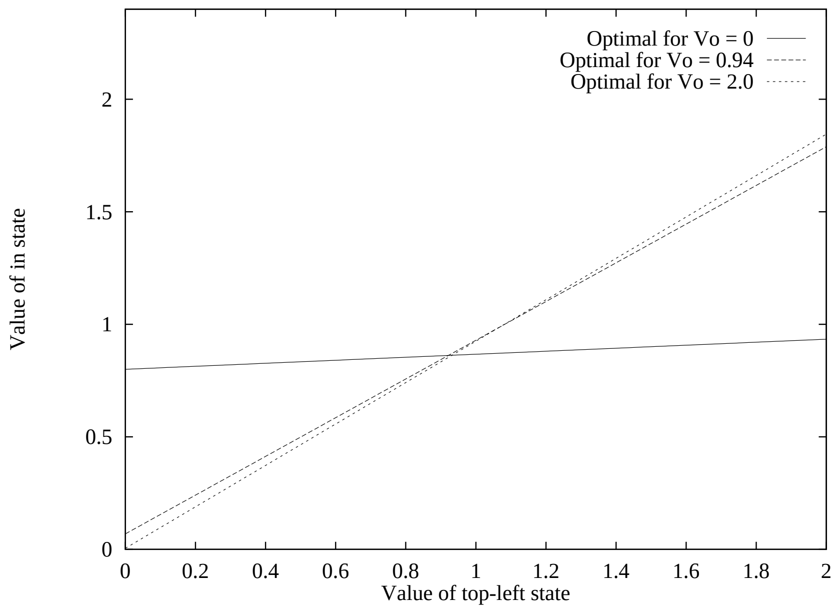

Figure 6.8: A third policy that is optimal at $\mathcal{V}^{\mathcal{O}} = [1.09]$ is added to the cache.

Note that the free variables in the system are the components of $\mathcal{V}^{\mathcal{O}}$. The objective function maximizes the Bellman error at state s under the assumption that action a is taken. The first set of constraints identifies the region in $\mathcal{V}^{\mathcal{O}}$ space for which π dominates at t. If this region exists, it is guaranteed to be a single, continuous facet of a convex surface by Lemma 3. The last set of constraints bounds $\mathcal{V}^{\mathcal{O}}$ to be within the range of possible values.

The largest value returned by the linear program over all s, a, and π provides the point in $\mathcal{V}^{\mathcal{O}}$ space at which the current cache of policies will have the largest Bellman error. The time bound is satisfied because linear programming is polynomial in the size of its inputs.■

This theorem provides a basis for determining if a policy cache is $\epsilon-$optimal. If the largest Bellman error returned by the linear program in the above theorem is less than ϵ, this means that no matter what values the states in the out-space of G assume, the policy assigned to G by a high-level SMDP using Π will produce state values with a Bellman error of less than ϵ. Thus, if the largest Bellman error is less than ϵ, the cache is ϵ-optimal. The computational consequences of Theorem 18 are non-trivial and deserve emphasis: When combined with Theorem 16, this appears to be the first efficient method known for determining if a set of policies for a region of an MDP is sufficient to produce a global solution that is within a bound of the optimal global solution *without considering the rest of the MDP*. Note that the conditions checked by Theorem 18 are sufficient, but have not been shown to be necessary; more efficient methods may be possible.

Assuming that the minimum possible state value is $V_{\min}$, this theorem provides a means of constructing an $\epsilon-$optimal cache of policies. Suppose that Theorem 18 is implemented as a function, find-worst, that takes a policy cache and returns two values, the point at which the Bellman error is worst, and the magnitude of this error.

```
$\Pi$ = {optimal policy for $V^{\mathcal{O}} = (V_{min}, \ldots, V_{min})$}
quit = false
Repeat until quit
  (worst-error, worst-point) = find-worst($\Pi$)
  if worst-error $> \epsilon$
    $\Pi = \Pi \cup$ {optimal policy for worst-point}
  else
    quit = true
```

This algorithm keeps adding policies to the policy cache until the policy cache is proven to be $\epsilon-$optimal. Each policy that is added covers at least one case where the current cache is inadequate. Note that, worst-point will always be at a corner of the convex facet defined by some dominating policy.

In practice, it is preferable to add a policy that is optimal for a point slightly towards the interior of the facet instead of at a corner. The reasoning for this is fairly subtle: If the new policy, π_2, is optimal at a corner, $\mathcal{V}^2$, of a facet of the region where an existing policy, π_1, dominates, and π_2 turns out to be only slightly better than π_1 at this corner, then the area in which π_1 will be optimal may remain nearly the same. In this case, the next call to find-worst, could return a $\mathcal{V}^3$ that is very close to $\mathcal{V}^2$, and in the worst case, the algorithm could produce a sequence of vertices, all very close to each other. By picking $\mathcal{V}^{2'}$ distance ϵ from $\mathcal{V}^2$ and towards the interior of the region where π_1 dominates, this ensures that any subsequent points returned by find-worst will be a distance of at least ϵ from the $\mathcal{V}^2$, avoiding the pathological case described above. The step towards the center of the dominating facet is shown in Figure 6.9.

The value space search algorithm also implicitly assumes that each new policy that is added to the cache will improve the value of every state in the region when $\mathcal{V}^{\mathcal{O}} =$ worst-point. This will be true if actions are sufficiently noisy such that reducing the Bellman error in any state will produce at least a minute improvement in the value of other states. It is possible to construct models where this assumption does not hold and in such cases additional constraints must be added to the linear program to break ties between policies. Suppose, for example, that when policy π_1 is adopted in state s_1, that the probability of reaching state s_2 is 0. If find-worst detects a large Bellman error for s_2 at point $\mathcal{V}^2$, the value space search algorithm could produce a new policy, π_2, that improves the value of s_2. However, this might not change the value of s_1 if π_2 still has probability 0 of reaching state s_2 when adopted from s_1. In other words, π_2 improves the policy in a part of the state space that is never reached from s_1 under π_1 and still is never reached under π_2. This is a problem for the value space search algorithm since the region of $\mathcal{V}^{\mathcal{O}}$ space over which π_1 dominates at s_1 will remain unchanged. Thus, find-worst will return the same $\mathcal{V}^2$ on subsequent calls, even if π_2 is added to Π.

One way around the problem of policies that improve the values of unreachable states is to alter the noise model so that every state has at least minute probability of reaching every other state. For any MDP, there will be some δ such that modifying the transition function so that $\mathcal{T}(s, a, s') \geq \delta$ will not change the optimal policy in any way. Unfortunately, this modification will remove any sparseness in the transition matrix and

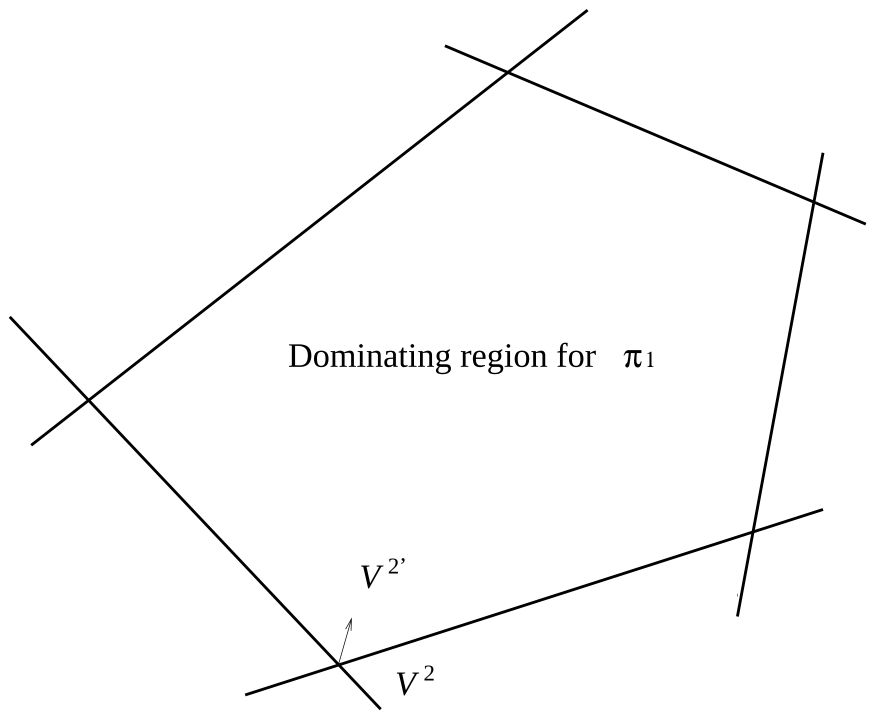

Figure 6.9: This figure shows value surfaces for a problem with a two state out-space. The surface is projected into two dimensions on the page, so the height off the page would correspond to the value of an in-space state. $\mathcal{V}^2$ is the point returned by a call to find-worst. It is better to generate a new policy for the point $\mathcal{V}^{2'}$ since the plane for this point will clip the corner of the dominating region for π_1, forcing subsequent calls to find-worst to pick points further away from $\mathcal{V}^2$.

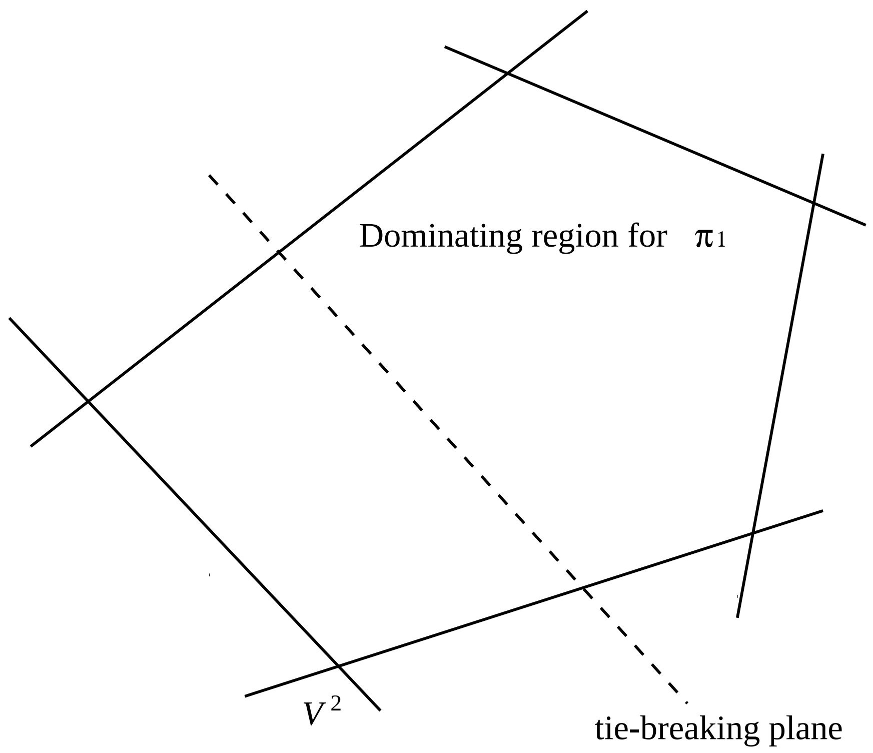

Figure 6.10: If two policies have the same value for an in-space state, but differ at the value of some other state, a tie-breaking plane is introduced to determine which policy is selected.

could increase significantly the cost of value iteration or policy iteration for these problems. Moreover, the introduction of the very small δ could introduce numerical stability problems for linear programming. A better approach is for the value space search algorithm to check each new policy before it is added to the cache. If π_1 previously dominated at state s_1 for value vector $\mathcal{V}^2$ and if π_2 is optimal for $\mathcal{V}^2$, but $f_{\pi_1}(s_1, \mathcal{V}^2) = f_{\pi_2}(s_1, \mathcal{V}^2)$, then π_2 should not be added to Π. If f_{π_1} and f_{π_2} are equal at s_1, then the dominating surfaces for these two policies are coextensive for s_1. However, π_1 and π_2 must differ at s_2, so the hyperplane defined by $f_{\pi_1}(s_2, \mathcal{V}^{\mathcal{O}}) = f_{\pi_2}(s_2, \mathcal{V}^{\mathcal{O}})$ can be used as a tie breaker. The condition $f_{\pi_1}(\mathcal{V}^{\mathcal{O}}, s_2) \leq f_{\pi_2}(\mathcal{V}^{\mathcal{O}}, s_2)$ would be added to the conditions defining the dominating region of π_2 in find-worst, and $f_{\pi_2}(\mathcal{V}^{\mathcal{O}}, s_2) \leq f_{\pi_1}(\mathcal{V}^{\mathcal{O}}, s_2)$ would be added to constraints defining the dominating region for π_1.

If the dominating region for the new policy, π_2, is not completely coextensive with the dominating region for π_1, then $f_{\pi_1}(s_1, \mathcal{V}^{\mathcal{O}}) = f_{\pi_2}(s_1, \mathcal{V}^{\mathcal{O}})$ is an edge in the dominating

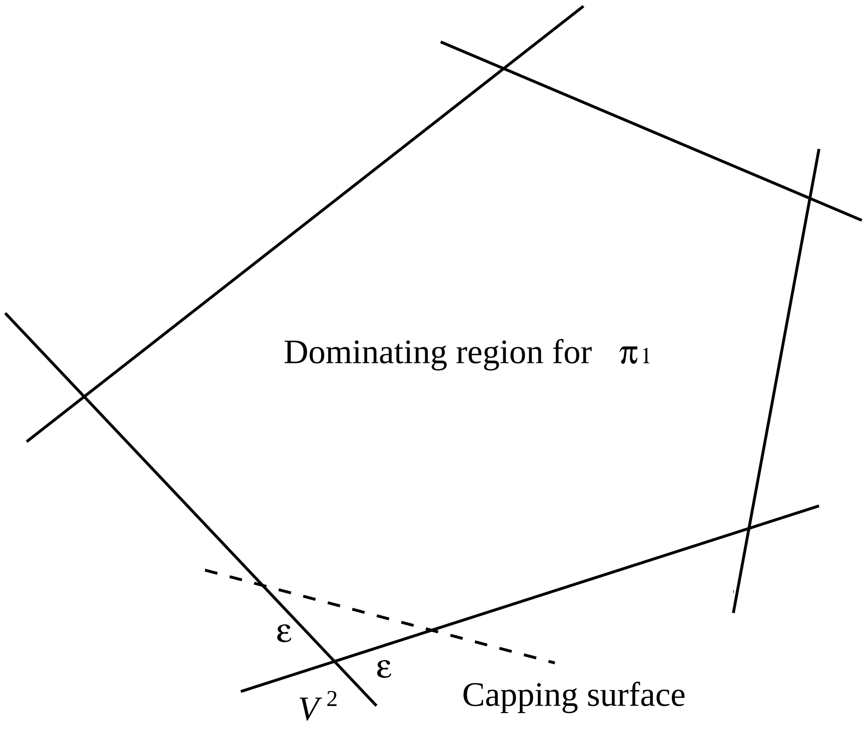

Figure 6.11: A "cap" is constructed around the point $\mathcal{V}^2$ by connect points at are ϵ away from $\mathcal{V}^2$ along the sides of the dominating region for π_1. The cap prevents find-worst from considering $\mathcal{V}^2$ on subsequent calls to find-worst.

region for π_1. In this case, adding a hyperplane to break ties between π_1 and π_2 is a little trickier. It can be done by selecting d of the facets that form the $\mathcal{V}^2$ vertex and picking points ϵ away from $\mathcal{V}^2$ on each facet in a direction that still borders the dominating region for π_1. The hyperplane connecting these $d+1$ points will partition the dominating region for π_1 by "capping" the corner containing $\mathcal{V}^2$. On the side of the hyperplane that contains $\mathcal{V}^2$, π_2 will be ϵ-optimal due to Theorem 17. This means that find-worst will need to search only the opposite side of the hyperplane and a constraint can be added to the linear program to enforce this.

The value space search algorithm has some similarities to and was inspired by algorithms for partially observable MDPs (POMDPs) (see Lovejoy (1991) for a survey), and in particular, the Witness algorithm (Cassandra, Kaelbling, & Littman, 1994). The treatment of policies as linear functions, the maximum over which forms a convex surface, is common in the POMDP literature. The approach used here can be seen as a multidimensional generalization of an observation made in Russell and Norvig (1995, Ex. 17.4). The value space search algorithm uses a similar approach to that of the Witness algorithm to search a continuous space to find the place where the error in the current set of policies is largest. The Witness algorithm is a synchronous value iteration algorithm that searches through belief space for a partially observable problem. The value space search algorithm searches through the space of state values to find the point at which the error in a set of infinite horizon MDP value functions is the largest.

The value space search algorithm was used to find an $\epsilon-$optimal policy cache for room 1 of Figure 6.1. This subproblem contains 25 states and has a fan-out of 2. The noise model was the same as the previous example, and the discount factor was 0.95. There are $4^{25} \approx 10^{14}$ possible policies for this subproblem. Of course, many of these are unreasonable policies that, for example, move the agent in circles. However, a variety of policies can still be induced by different values of the out-space states, even in such a simple problem. One would think that the agent would simply aim for the exit with the highest state value. However, the noise in the action model ensures that there is always some chance that the agent will wind up unintentionally exiting the wrong way. Thus, as the relative difference between the two out-space states increases, the optimal policy will take a more circuitous route towards the desired exit, hugging the walls to avoid accidentally getting too close to the undesired means of egress.

If the values of the states are assumed to be on $[0 \ldots 20]$, then the ϵ-grid approach for this problem would require 4 *million* policies for $\epsilon = 0.01$. The value space search algorithm produced a policy cache with the same optimality guarantees with just 22 policies. For $\epsilon = 0.001$, the ϵ-grid approach would require 400 *million* policies, while the value space search algorithm produced the same 22 policies.

In this particular case, the value space search algorithm has captured the intuition that this type of subproblem should not be that hard. A few seconds of computation has produced a small cache of that will ensure a nearly optimal solution for this region no matter what happens in any connecting region. This small subproblem is now decoupled and completely solved — at least for $\epsilon \geq 0.001$ and for problems where the neighboring states can assume values on $[0 \ldots 20]$. Any MDP satisfying these conditions and with an optimality requirement of no more than $\frac{0.001}{1-0.95} = 0.02$ will *never* need another policy defined on this region.

The value space search algorithm solved a 9×9 room with two exits using just 60 policies, but several minutes of CPU time. The additional time is largely attributable to an inefficient lisp-based linear program solver. The very encouraging aspect of this result is that the number of policy policies grew more slowly than the number of states.

An important unanswered question is whether the value space search algorithm is guaranteed to find a polynomial size $\epsilon-$optimal cache of policies if such a cache exists. The idea of creating a new policy near the point where the current policy cache performs the worst is plausible, but there is not yet a proof that this constructs a cache of policies that is in any way minimal. A drawback of this algorithm is that it solves a large number of linear programs. This can be onerous if the number of states in the region is large. Of course, this price is paid only once, and the cache can be reused indefinitely in any MDP that contains the same subproblem. Moreover, many implementation tricks can be used to reduce the size and number of linear programs constructed. For example, the maximum Bellman error for any (entry-point,state,action,policy) quadruple is non-increasing as the policy-cache grows, so the solutions to previous linear programs can be cached across calls to find-worst. A new linear program is needed only if the cached error is greater than the maximum error detected so far in the current call to find-worst.

6.3.3 The Convex Hull Bounding Approach

This section sketches a third algorithm with a computational geometry flavor. This algorithm also provides slightly different optimality guarantees than the previous algorithms. It guarantees that a cache of policies will be ϵ-optimal *at the high-level.* Recall that actions in the high-level SMDP correspond to policies at the low level. If a cache is $\epsilon-$optimal at the high level, this means that there is no low-level policy that could improve the value of a high-level state by more than ϵ. In the four-room example, this would mean that no policy could improve the value of one of the connecting states by more than ϵ. However, there could be a policy that could improve the value of some other state, for example, the state in the top left corner of the model, by more than ϵ. The expected effect of this change on any of the high-level states must be less than ϵ.

High-level ϵ-optimality implies that any policy starting from a high-level state (e.g. one of the states connecting the rooms) will have an expected value within $\frac{\epsilon}{1-\beta}$ of optimal. This could be a problem, however, if the agent typically starts in some state that is not a high-level state. In such cases, the starting position of the agent can be treated as if it were a connecting state by adding it to the in-space of the enclosing region and constructing a policy cache as if it were a connecting state. If desired, every state could be treated as if it were an in-space state, ensuring full low-level optimality as well.

The algorithm presented in this section has run time that is exponential in d, the fan-out of the region, but unlike the ϵ-grid approach, it does not depend explicitly on $1/\epsilon$ and unlike the value space search algorithm, it can avoid considering every state inside of a region if high-level ϵ-optimality is sufficient. The algorithm relies upon the following formal results:

Lemma 4 *For any point $\mathcal{V}^{\mathcal{O}}$, set of points, $\mathcal{V}_1 \ldots \mathcal{V}_{d+1}$, with set of policies, $\pi_1 \ldots \pi_{d+1}$, such that π_i is optimal with respect to $\mathcal{V}_i$ and such that the $\mathcal{V}_i$ form a convex hull around $\mathcal{V}^{\mathcal{O}}$, the optimal policy with respect to $\mathcal{V}^{\mathcal{O}}$ at any state s is bounded from below by $\max_i f_{\pi_i}(s, \mathcal{V}^{\mathcal{O}})$ and from above by the hyperplane containing each of the $(\mathcal{V}_i, f_{\pi_i}(s, \mathcal{V}_i))$.*

Proof: Bounding from below is obvious and follows from Lemma 3: the optimal policy at any point must do at least as well as the dominating policy in the cache. The bound from above is somewhat more subtle: Let H_1 be the hyperplane containing the $(\mathcal{V}_i, f_{\pi_i}(s, \mathcal{V}_i))$. Suppose that there exists some π and corresponding f_π such that for some s, $f_\pi(s, \mathcal{V}^{\mathcal{O}})$ is above H_1. Let H_2 be the hyperplane corresponding to the linear value function of this policy at s. There must exist some corner of the convex hull used to create H_1 (some $(\mathcal{V}_i, f_{\pi_i}(s, \mathcal{V}_i))$) where H_2 is above H_1, i.e., $f_\pi(s, \mathcal{V}_i) > f_{\pi_i}(s, \mathcal{V}_i)$. However, f_{π_i} is known to be optimal with respect to $\mathcal{V}_i$, so this is a contradiction.■

A simple example of this lemma is shown in Figure 6.12. Two functions for policies from the single-exit room are shown. One is optimal at $\mathcal{V}^{\mathcal{O}} = 0$ and the other is optimal at $V^{\mathcal{O}} = [2]$. For any $0 \leq \mathcal{V}^{\mathcal{O}} \leq 2$, the linear function for the optimal policy cannot cross 0.8 at $\mathcal{V}^{\mathcal{O}} = [0]$ or cross 1.84 at $\mathcal{V}^{\mathcal{O}} = [1.84]$. Thus, the value of optimal policy is bounded by the line shown.

Theorem 19 *For region G and cache of policies, $\pi_1 \ldots \pi_m$, that are optimal at $\mathcal{V}_1 \ldots \mathcal{V}_m$, the optimal policy value for any s with respect to any $\mathcal{V}^{\mathcal{O}}$ is bounded from below by the convex surface formed by the maximum over the corresponding $f_{\pi_1} \ldots f_{\pi_m}$ and bounded from above by the convex hull containing the points: $(\mathcal{V}_1, f_{\pi_1}(s, \mathcal{V}_1)), \ldots, (\mathcal{V}_m, f_{\pi_m}(s, \mathcal{V}_m))$.*

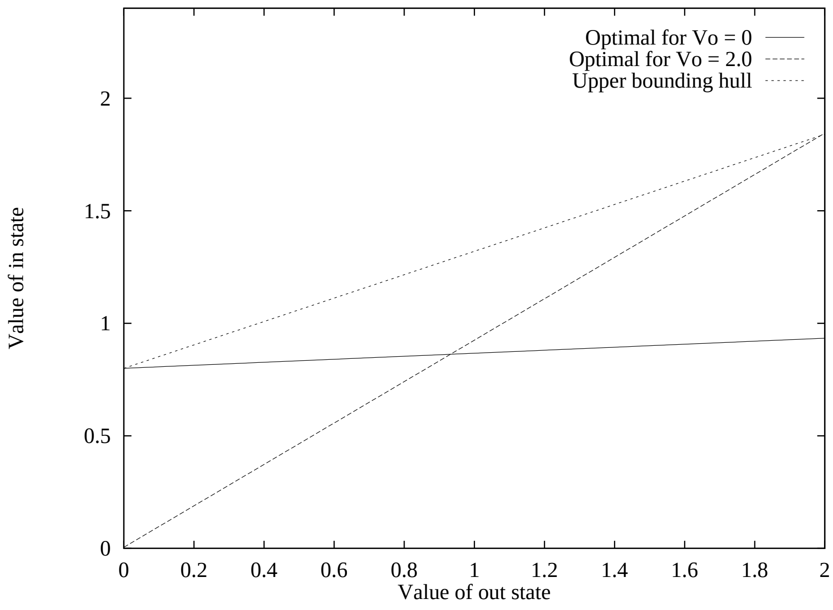

Figure 6.12: Two policies, and an upper surface bounding their distance from optimality.

Proof: The bound from below is a direct consequence of Lemma 3. The bound from above follows from Lemma 4 and noting that the lowest bounding hyperplane for any $\mathcal{V}^{\mathcal{O}}$ must form a facet in the convex hull of $(\mathcal{V}_1, f_{\pi_1}(s,\mathcal{V}_1)), \ldots (\mathcal{V}_m, f_{\pi_m}(s,\mathcal{V}_m))$.■

This theorem also suggests an algorithm for finding points in value space where the current policy cache is not $\epsilon-$optimal: For each facet in the upper bounding hull, find the point in the lower hull that maximizes the distance between the two surfaces. If no point can be found where the distance is greater than ϵ, then the policy cache is ϵ-optimal. If the facets in the upper hull are enumerated as a set of linear functions, $g_1 \ldots g_l$, then the maximum distance can be checked for each s, f_{π_i}, and g_j as follows:

Maximize

$$g_j(s,\mathcal{V}^{\mathcal{O}}) - f_{\pi_i}(s,\mathcal{V}^{\mathcal{O}})$$

Subject to

$$\begin{aligned} g_j(s,\mathcal{V}^{\mathcal{O}}) &\leq g_k(s,\mathcal{V}^{\mathcal{O}})\ \forall k \\ f_{\pi_k}(s,\mathcal{V}^{\mathcal{O}}) &\leq f_{\pi_i}(s,\mathcal{V}^{\mathcal{O}})\ \forall k \\ \mathcal{V}^{\mathcal{O}}[k] &\leq V_{\max}\ \forall k \end{aligned}$$

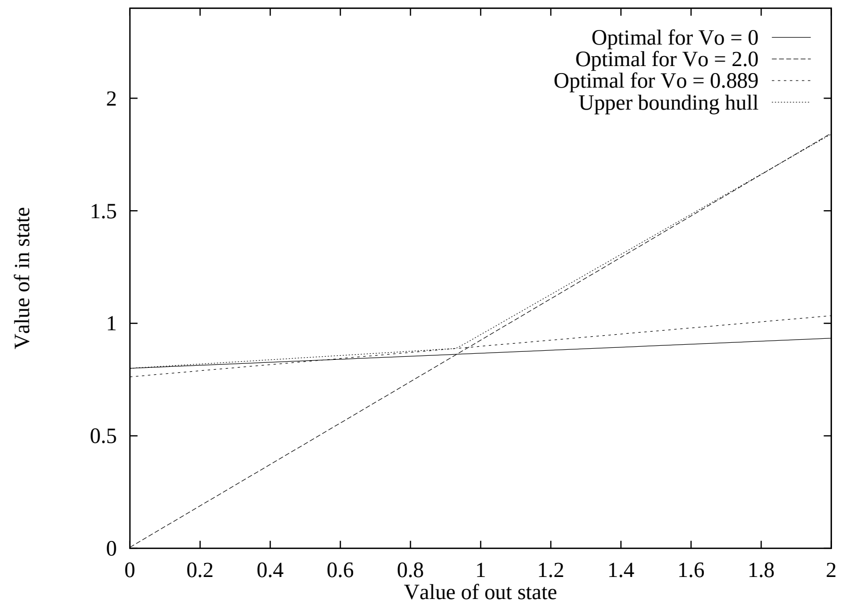

Figure 6.13: Adding a new policy that is optimal for $\mathcal{V}^{\mathcal{O}} = [0.932]$ pulls the upper-bounding hull down, dramatically tightening the bounds on the value of any optimal policy on $0 \leq V^{\mathcal{O}} \leq 2$. The upper hull is hard to see because it is very close to the lower surface. This means that three policies shown are nearly optimal for the range shown.

The last constraint bounds $\mathcal{V}^{\mathcal{O}}$ to lie in the permitted range. The first two sets of constraints identify the area in $\mathcal{V}^{\mathcal{O}}$ space in which f_{π_i} is a facet on the lower bounding hull *and* g_j is a facet on the upper bounding hull. If such an area exists, the objective function finds the point at which the distance from the upper hull to the lower hull is greatest.

The above linear program can be used to generate a cache of policies in a fashion similar to the value space search algorithm. By searching all pairs of upper bounding facets and lower bounding facets, the point at which the gap between these surfaces is greatest can be used to determine a new policy. Consider again the two policies in Figure 6.12. The distance from the upper hull to the lower hull is largest when $\mathcal{V}^{\mathcal{O}} = [0.932]$. The optimal policy for this region when $\mathcal{V}^{\mathcal{O}} = [0.932]$ assigns the top-left state a value of $0.136\mathcal{V}^{\mathcal{O}} + 0.762$. The upper hull containing this point is shown in Figure 6.13. The distance between the upper bounding hull and the lower hull is now quite small, indicating that these three policies are nearly optimal if the agent starts in the top-left corner and the value of the out-space state is on $[0 \ldots 2]$.

One complication for the bounding hull approach is that the upper-bounding hull may not cover the entire space of values for $\mathcal{V}^{\mathcal{O}}$. For points outside the hull, the value of the optimal policy can be bounded from above by $V^{\max}$. In some cases this can be tightened by observing that no policy will do better than the sum of value of the optimal policy at $(V^{\min}, \ldots, V^{\min})$, and $\beta \max_i \mathcal{V}^{\mathcal{O}}[i]$, since the optimal policy for the lowest values of the out-space states will maximize the reward received within the region, and no policy can do better than receiving this reward and then moving to the highest valued state in one step.

The more serious complication for this algorithm is the general result from computational geometry that the convex hull of m points in d dimensional space can have $O(m^{\lfloor \frac{d}{2} \rfloor})$ facets, making this algorithm exponential in d. Still, the convex hull bounding algorithm is superior to the $\epsilon-$grid approach since the $\epsilon-$grid approach has run time that depends directly on $\frac{1}{\epsilon}$ and the range of values possible in the out-space, while the bounding approach depends on the number of policies in the cache.

6.4 Partial Decoupling

The previous section presented three algorithms for completely decoupling MDPs. These algorithms are quite computationally intensive, and there are no guarantees that the size of the policy cache required for a desired solution quality will be manageably small. In such cases, one may be forced to use a policy cache that is not known *a priori* to be $\epsilon-$optimal for the range of values the out-space of a region will take on when reconnected to the rest of the MDP. For example, a rough policy cache could be constructed for each room of a large navigation problem. When a high-level SMDP that combines these rooms is solved, some decisions will need to be made on-the-fly about whether the policies in the rough cache are adequate for the larger problem.

More specifically, suppose an MDP has been divided into disjoint regions and a policy cache has been constructed for each region. A high-level SMDP problem can be defined over the out-spaces of these regions. For a particular region, G, an algorithm solving this high-level decision problem has the option of using one of the policies in the policy cache for G, or generating a new policy that is optimal for the algorithm's current estimate of $\mathcal{V}^{\mathcal{O}}_G$. A straightforward way to answer this question would be to use the cached f_π functions to assign values to every state in the problem and then compute the Bellman error for each state. However, this approach would require so much computation that it essentially would defeat the purpose of solving a high-level problem. Instead, high-level optimality can be checked quite efficiently by using the tools of the convex hull bounding algorithm.

Starting with some policy cache, $\pi_1 \dots \pi_m$, the elements of which are optimal at the corresponding $\mathcal{V}_1 \dots \mathcal{V}_m$, for any particular $\mathcal{V}_G^{\mathcal{O}}$, the value of any state under the optimal policy with respect to $\mathcal{V}_G^{\mathcal{O}}$ is bounded from below by $\max_{\pi_i} f_{\pi_i}(s, \mathcal{V}^{\mathcal{O}})$, and the value is bounded from above the convex hull formed by $(\mathcal{V}_1, f_{\pi_1}(s, \mathcal{V}_1)) \dots (\mathcal{V}_m, f_{\pi_m}(s, \mathcal{V}_m))$ (Theorem 19). The situation here is slightly different from the bounding algorithm in that $\mathcal{V}_G^{\mathcal{O}}$ is fixed and known. Instead of a high-dimensional convex hull problem, the bounds for a *particular* $V^{\mathcal{O}}$ can be determined by solving a linear program. In the following f_* is an unknown linear equation, i.e., the coefficients and constant are free variables:

Maximize:

$$f_*(s, \mathcal{V}^{\mathcal{O}})$$

Subject to:

$$\begin{aligned} f_*(s, \mathcal{V}_i) &\leq f_{\pi_i}(s, \mathcal{V}_i), 1 \leq i \leq m \\ f_*(s, \mathcal{V}_{\mathcal{O}}) &\leq V_{max} \end{aligned}$$

To reassure oneself that this is indeed a *linear* program, recall that in this context, $\mathcal{V}^{\mathcal{O}}$, the $\mathcal{V}_i$, and coefficients and constants for the f_{π_i} are all known constants. The only variables are the components of f_*. The first set of constraints requires that f_* be no better than the optimal policy for s at points in value space where the optimal policy is known. This is, essentially, a restatement of Lemma 4. The second set of constraints requires that f_* never exceeds the maximum value any state can assume in this problem. Thus, the objective function forces the linear program to find the highest hyperplane that does not violate Lemma 4 or the bound on state values. If $\mathcal{V}^{\mathcal{O}}$ lies in the convex hull of $\mathcal{V}_1 \dots \mathcal{V}_m$, then f_* will be the facet of the upper-bounding convex hull from Theorem 19. Note that if $\mathcal{V}^{\mathcal{O}}$ does not lie in the convex hull, $V_{\max}$ will be returned. This bound can be tightened by requiring that the constant of f_* be no larger than the value of the optimal policy at $\mathcal{V}^{\mathcal{O}} = (V_{\min} \dots V_{\min})$ and that the coefficients of f_* sum to be no more than 1.

If the distance between the dominating policy and the upper bound returned by the above linear program is less than ϵ for every state in the in-space of G, then the policy cache for G is sufficient to produce a high-level ϵ-optimal policy for the current value of $\mathcal{V}^{\mathcal{O}}$. This means that a high-level decision problem can, for now, avoid updating the policy for region G and focus attention on other regions. This decision will need to be reevaluated as values of the states in the out-space of G change. One way to view this result is that it enables a form of high-level prioritized sweeping (Moore & Atkeson, 1993; Andre et al., 1997).

This result also has significant consequences for the transfer of knowledge across problems. Suppose, for example, that a particular model substructure appears in many different problems. Consider a larger version of the four-room problem with many interconnected rooms. Different tasks in this domain would correspond to different positions of the reward in different rooms. Every time a policy is produced for a room it can be added to the room's policy cache. The above linear program can be used to determine quickly if for some new problem, the cache in a particular room is adequate. Thus, a form of cross-task learning is achieved where the time required to plan for new objectives declines as experience is gained with the environment. Moreover, intelligent allocation of computational resources will be possible since parts of the value space that have already been mastered will no longer drain computational resources.

6.5 Hierarchical Decomposition

Suppose that the four-room example is just part of a larger sixteen-room problem, as shown in Figure 6.14. This problem is just four large rooms each of which is composed of four smaller rooms. If an ϵ-optimal policy cache is devised for each small room, the policies in the caches will become actions for an SMDP defined at the level of the enclosing large room. An $\epsilon-$optimal policy cache for the large rooms can be constructed using the solutions from the small rooms as actions. These policies can be combined to produce a global policy that is $\frac{\epsilon}{(1-\gamma)^2}$-optimal.

In general, if an MDP can be divided into nested sets of disjoint subproblems k levels deep, and if ϵ-optimal solution caches are constructed at each level such that cached policies for level $j+1$ become actions for level j, then the overall solution will be $\frac{\epsilon}{(1-\gamma)^k}$ optimal.

6.6 Application to HAMs

The decomposition algorithms of this chapter can be applied to SMDPs induced by HAMs in a very natural way. Recall the hallway navigation problem from the previous

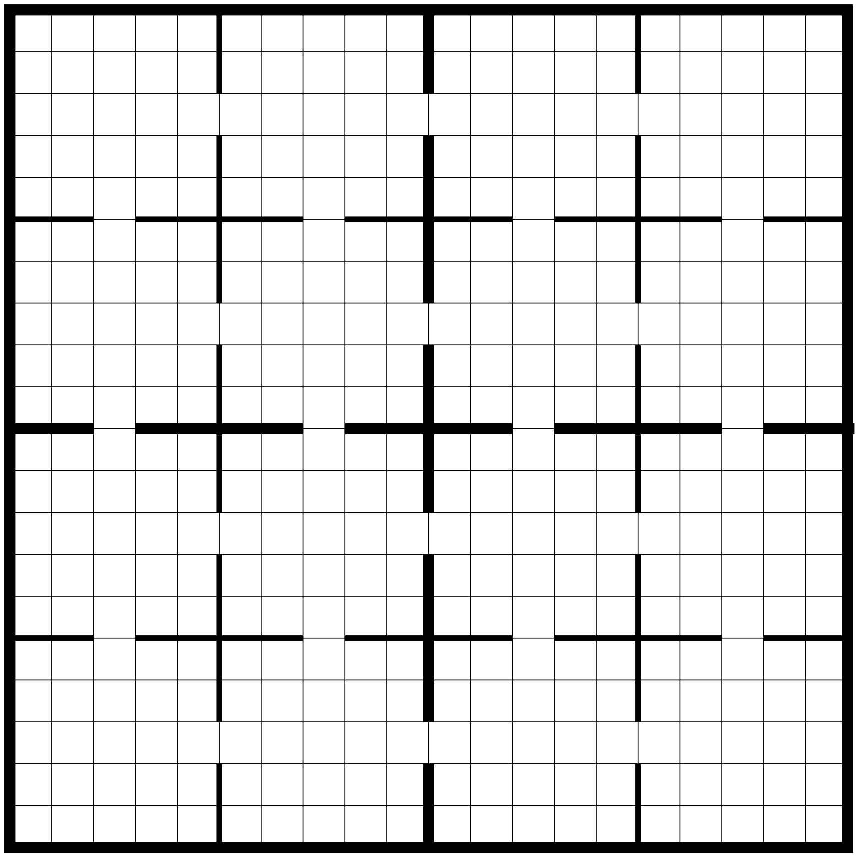

Figure 6.14: A sixteen-room problem that contains four nested four-room problems.

chapter. In this problem, a navigation HAM, $\mathcal{H}$, was combined with a large MDP, $\mathcal{M}$, to produce a reduced SMDP, $reduce(\mathcal{H} \circ \mathcal{M})$. In this SMDP, the states can be partitioned into regions based upon the machine in the HAM that is running at each state. For example, in the top hallway in Figure 5.10, there will be a collection of connected states where the machine for moving right across the hallway is running. This machine will stop if the agent hits either end of the hallway and the high-pointing east or west sonar is active. $\mathcal{V}^{\mathcal{O}}$ for this problem will be a vector of size 24, with 12 state strips on each end of the hallway.

One of the nice synergies between HAMs and the algorithms in this chapter is that a well-crafted machine hierarchy also will induce weak coupling between subproblems for problem decomposition algorithms. A well-designed HAM will aim to minimize the number of choice points in the induced SMDP, minimizing the complexity of optimally refining the HAM for a new task. If the HAM call structure is used to induce a hierarchical partitioning for one of the decoupling algorithms in this chapter, the number of choice points induced by the HAM at each level of the HAM will determine the number of states in the SMDP at each level of the decomposition. This determines size of the out-space for each subproblem.

The navigation HAM from Chapter 5 is an example of this synergy. One of the reasons why the HAM is so successful is that it limits the number of times the agent can pick a new hallway direction machine. This limits the number of choice points, but it also limits the size of the out-space for a subproblem of moving across a hallway since the choice points for the HAM become out-space states for a decomposition algorithm.

6.7 Application to Reinforcement Learning

It would be difficult to apply the results from this chapter directly to reinforcement learning since they rely heavily upon knowledge of the underlying model dynamics. However, there are two ways in which a reinforcement learning agent can benefit decoupling theorems in this chapter. If the agent knows that a new problem will share some structure with a previous problem, the agent can use cached policies from previous experiences with these subproblems. The value functions for these cached policies can be learned on-line using the symbolic TD algorithm from Chapter 4. The agent could use these to determine which areas of the problem requires new policies and could attempt to focus its exploration in these areas.

A second option for a reinforcement learning agent would be to make several guesses about the ultimate values of the out-space states for a particular subproblem and learn optimal policies for these guesses as parallel reinforcement learning tasks (Chapter 4). If the subproblem is smaller than the overall problem, the agent will be able to learn policies and linear value functions for these policies in less time than it would take to learn a good global policy. The agent could then use the algorithms from this chapter to determine if some combination of the subproblem solutions will produce a satisfactory global solution. The agent also could use this information to influence its exploration strategy.

6.8 Conclusion

This chapter presented two approaches to decoupling MDPs, a complete decoupling approach and a partial decoupling approach. With complete decoupling, the problem is divided into independent subproblems, and the solutions to these subproblems are combined by solving a smaller SMDP. Two new algorithms for determining $\epsilon-$optimal policy caches for a subproblem are presented. The significance of the first algorithm is that it uses a polynomial time test to determine when to add a new policy to a cache. The second algorithm uses a computational geometry approach that can be exponential in the fan-out of the subproblem, but can be more efficient than the first algorithm if the fan-out is small.

Since complete decoupling may not always be possible, a method for partial decoupling is presented. This method assumes that an imperfect policy cache is used by a high-level asynchronous MDP algorithm. It uses the policy cache to bound the optimal values of states in a region with respect to the values of the states in the out-space of the region. By providing upper and lower bounds, this permits intelligent decisions about when to update the policy cache for a region based upon the algorithm's current estimate of the values of the states in the out-space of the region.

Together these results provide a framework for large-scale parallelization of MDPs and a formal framework for the transfer of knowledge across problems that share common structures. These results can be applied hierarchically, and the structure of a HAM can be used to determine the problem decomposition.

Chapter 7

Conclusions and Future Work

This chapter sketches some of the exciting areas for future research that have been opened up by the line of research in this dissertation and summarizes its contributions.

7.1 Future Work

The HAM language alone provides fertile ground for future research. The basic HAM language was designed to be as simple as possible, with the goal of making it amenable to analysis. Now that the basic concepts and formal properties have been established, this section provides some extensions to the HAM language, many of which are syntactic sugar that can make the language friendlier to the user.

With a firm grasp on the relationship between HAMs, temporal abstraction and hierarchical decomposition, it is now appropriate to consider the relationship between these topics and the seemingly orthogonal issues of value function approximation and state space abstraction. There is some reason to believe that these issues may not be completely orthogonal and that there could be important synergies. Finally, this section offers some thoughts on the problem of partial observability.

7.1.1 Extending HAMs

There are many possible extensions to the basic HAM language. This subsection describes some of the more promising possibilities.

Annotating States with Heuristic Information

One of the interesting characteristics of the HAM language, separating it from other attempts to introduce hierarchy into the MDP framework, is that it does not require any guesses about the values of states; HAMs encode procedural knowledge. However, there is no reason why the language could not be augmented to include guesses about state values. These could be used to construct an initial value function for the induced SMDP, or they could provide initial $\mathcal{V}^{\mathcal{O}}$ vectors for problem decomposition. Choice states also could be augmented to include suggestions about a default action and these suggestions could be used to create an initial policy for the induced SMDP. These heuristics would not be binding in any way and would not change the optimal policy for the induced SMDP. They simply would provide a heuristic starting point for an SMDP solution to the reduced model.

Parameter Passing and Return Arguments

Two other natural extensions to the HAM language are the introduction of parameter passing and return values for subroutine calls. If the parameters and returned values are constrained to be from some finite alphabet, then all of the theorems and algorithms from Chapter 5 will apply directly since these extensions will maintain the finite-state aspects of the language. With these extensions, machine subroutines could indicate *why* they have returned. For example, a hallway navigation machine could return a value distinguishing between stops that occur at the end of hallways, and stops that occur because the agent has slipped backwards into the previous intersection.

Stochastic, Adjustable Choice Points

One avenue that Chapter 5 explicitly avoided was the use of continuous or countably infinite state machine representations. One interesting step in this direction is the introduction of *adjustable, stochastic choice points.* The idea behind a stochastic choice point is that instead of requiring a hard choice between several alternatives, a refinement for the HAM can specify a probability distribution over alternative next machine states. If a distinct probability distribution were allowed for each choice point, then this would not be a particularly interesting extension since the optimal policy for the induced SMDP will always be deterministic. However, if the choices for all choice points induced by a particular machine state are chosen from the same distribution, then the problem becomes one of finding the optimal probability distribution that will be shared across a large number of choice points.

Some methods do exist for tuning stochastic action choices in policies that map from states to actions (Jaakola, Singh, & Jordan, 1995). The question of tuning a global parameter for a *machine* is important because it may help reduce the complexity of refining a machine for a new task. Suppose, for example, that a machine is intended for use in a symmetric room. An optimal decision at one side of the room may constrain the optimal decision at the other end. If there were a way for a single parameter to represent both of these choices, the complexity of the reduced SMDP for this problem could be cut in half. Achieving this savings in practice may be difficult, but there are some interesting possibilities. Suppose machine $\mathcal{N}$ contains a choice state m that chooses next state n with probability p. The parameter p will affect the next state choice for any extended state that contains m as a component. How could p be adjusted to improve $\mathcal{N}$'s performance over some region of the state space? The value of any state that launches $\mathcal{N}$ will be a function of p, but the difficulty is that unlike regular choice points, the function will not be linear.

The problem of choosing an optimal, global, continuous parameter for a machine can be framed as an SMDP. However, the max in the Bellman equation would no longer be a maximization over a finite set of discrete choices, but would require finding the maximum of a continuous non-linear function. Gradient descent methods could be used to find at least a local maximum in such cases, and local maxima may suffice for many problems. It remains to be seen if the flexibility afforded by continuous, adjustable parameters of this type justifies the computational cost of constructing and maximizing a non-linear function. Recent results in Marbach and Tsitsiklis (1998) may be applicable to this problem.

More Advanced Machine Representations

While finite state automata are a natural and frequently used representation for controllers, they are not as powerful or general as push-down automata or Turing machines. In principle, the HAM theorems all generalize to these machine representations if the contents of the stack or machine tape are considered part of the overall machine state. The difficulty with these representations is that the number of possible machine states becomes countably infinite, making the induced SMDP countably infinite as well. This makes it practically impossible to construct the reduced state space for these cases since the full state space cannot be manipulated directly. One alternative is the use of reinforcement learning methods. Since HAMQ-learning does not directly manipulate the full state space, if the stack depth (or number of Turing machine tape cells used) is finite, then HAMQ-learning can use a hash table to store values for the states that actually occur. This application of HAMQ-learning will be guaranteed to learn the optimal refinement of HAMs represented as Turing machines.[1]

Inventing New Machines

An intriguing area for future research is the automated discovery of good machine structures for HAMs. One way to view this issue is as an inductive learning problem. In constructing a HAM, one must have in mind some class of MDPs, or regions thereof, to which the HAM would be applied. Optimal solutions to MDPs from the class would be instances for the inductive learner. The hypothesis class would be the space of HAMs and the target hypothesis would be a HAM that optimizes some tradeoff between performance and the description length of the induced SMDP. Description length was used as a metric for the acquisition of general skills in Thrun and Schwartz (1995).

Another approach to the automatic generation of HAMs is the use of plans generated by classical planners as the basis for a machine structure. This approach was explored by Lin (1997), where a classical planner capable of producing plans with loops was proposed as a means of generating a type of abstract MDP policy. This approach was primarily focused on tasks of achievement that had specific decomposition properties, but it may be possible to combine these ideas with the HAMs.

Feudal Reinforcement Learning (Dayan & Hinton, 1993) and MAXQ value function decomposition (Dietterich, 1997) use a task hierarchy to generate abstract actions for reinforcement learning automatically. These methods use a pre-specified task hierarchy to make guesses about the value of reaching certain states in a manner similar to the method used by Lin (1993) to generate robot subtasks. These methods also use a form of state aggregation to generalize information about achieving abstract subtasks in one part of the state space to other parts of the state space. These methods are intriguing and have the potential to produce good practical results in some cases. One concern, however, is that

[1]Of course, if a Turing machine is guaranteed to use a finite number of tape cells, then there must be an equivalent finite state machine. However, determining if a Turing machine uses a finite number of cells *a priori* may be undecidable.

due to their use of state aggregation, it is difficult to make strong formal claims about the performance of these methods. One interesting possibility for future work would be some combination of these methods with HAMs in a way that could still preserve some of the formal guarantees of HAMs.

7.1.2 State Aggregation

As indicated in the beginning of the dissertation, state aggregation is an important method that is used at some level in almost any successful application of MDP methods. There are many important questions in state aggregation that remain unresolved and for this reason the topic was not addressed explicitly to avoid confusing the presentation and complicating the analysis of the methods presented here.

One of the unsatisfying aspects of the more formally justifiable state aggregation methods is that they fail to capture much of the intuitive notion of an abstract state. For example, the approach used in Dean et al. (1997) offers formal guarantees on the relationship between the solution to an abstract problem and a solution to the original problem based upon the differences between the reward and transition functions of the underlying states comprising each aggregated state. If two states with vastly different transition functions are aggregated, these methods cannot guarantee a close relationship between the solution to the aggregated model and the original model. The problem with this is that very frequently in human reasoning it seems quite natural to aggregate states that are very different. For example, the states in a room of one's house can have very different transition functions — moving forward from one state will exit the room, while moving forward from another will bump into the television. In spite of these differences, people seem to reason about objects like rooms as if they were single states. People plan to go "to the living room," and treat the living room as a discrete place at a certain level of abstraction. The state aggregation methods of Dayan and Hinton (1993) and Dietterich (1997) attempt to capture these intuitions, as do some methods proposed in Dean and Lin (1995). However, these methods do not provide compelling formal guarantees.

State aggregation can be applied to HAMs in a straightforward way, by applying state aggregation methods to a HAM-induced SMDP. This will be subject to the same complications and pitfalls of ordinary state aggregation. A more promising avenue for exploiting state aggregation methods is in combination with the decomposition algorithms of

Chapter 6. First, state aggregation can be used combine similar out-space or in-space states for different regions of an MDP. For example, if the region of physical space corresponding to the threshold of a door were modeled as several discrete states, these states could be aggregated with little loss in performance guarantees because they are so similar. This would improve the performance of decomposition algorithms, which are heavily influenced by the size of the out-spaces for subproblems. After a cache of policies is constructed for each subproblem, it may be possible to aggregate different in-space states, even those that were originally very far apart. This is because the intermediate states essentially have been removed from the state space through the introduction of a policy cache. Thus, states that previously had no direct connection would now have direct connections to each other, permitting state aggregation.

7.1.3 Function Approximation

Function approximation can be incorporated directly into any of the methods described in this dissertation. For example, a function approximator could be used for the value function of a HAM-induced SMDP in exactly the same way function approximation is used for ordinary MDPs. If the HAM-induced SMDP is very large, function approximation can be used as part of the state elimination process in combination with symbolic value determination (see Chapter 4).

Another possible role for function approximation with HAMs is their use as a means of generalizing the value of launching HAM subroutines across different parts of the state space or across different problems. For example, a neural network could be used to generalize the value of starting the machine from Chapter 5 for moving right across a hallway. In general the value of starting this machine will be a linear function of the values of the states in which the machine terminates. The coefficients of this function will depend upon a number of factors such as the length and width of the hallway, and the number of obstacles in the hallway. A neural network using these features as inputs could be trained to produce a good estimate of the effects of this particular machine based upon experience with similar hallways. In principle, the estimates from the neural network could replace the costly state removal algorithms from Chapter 4. In the case of reinforcement learning, symbolic TD could be used with function approximation to learn symbolic representations of the value of different subroutines. These representations could be stored and used by the agent to get quick estimates of the value of a subroutine call in a new situation. This stored, generalized experience could reduce the amount of trial and error necessary to learn in new environments.

The danger of function approximation for HAMs is that unless a very conservative approximation method is used, such as Gordon's Averagers (Gordon, 1995), all of the formal guarantees of the HAM method will be lost. Since function approximation is often used in practical problems in spite of the loss of formal guarantees, the combination of function approximation and HAMs may still be worthwhile. Moreover, since function approximators, as used with HAMs, would need to represent values for only a portion of the state space, function approximation methods should behave more stably. In the case of reinforcement learning, they should learn faster and generalize better.

Function approximation also can play a role in the decomposition algorithms of Chapter 6. It can be used to produce approximate solutions to large subproblems, or as a means of generalizing the effects of policies in a solution cache to different subproblems. Unfortunately, the pitfalls of function approximation are even more of a problem for decomposition algorithms, which rely on precise computations of the value relationships between states to establish tight performance bounds on policy caches. The errors resulting from function approximation could be magnified by the decomposition algorithms. Still, this combination could be worth pursuing since the performance of function approximation on practical problems is often much better than worst-case analyses would suggest. Furthermore, it may be possible to use distribution assumptions about the space from which subproblems are drawn to obtain error bounds.

7.1.4 Partial Observability

Partial observability refers to the case where an agent does not have complete knowledge of the state of the environment. Instead, the agent may have some partial information that gives a clue about the underlying state, but that is not sufficient to identify the current state of the environment with certainty. The problems are Partially Observable MDPs (POMDPs). In POMDPs, an agent generally is forced to make decisions based upon either a complete history of its actions and perceptions in the environment, or a probability distribution over the states in the environment called a *belief state* or *information state.*

The history-based approach to acting in POMPDs is problematic unless the environment belongs to a special subclass of environments for which a bounded *history window* is a sufficient basis for decision making. Belief state representations are also problematic since planning with such representations is extremely difficult. POMDPs are known to be PSPACE-hard (Papadimitriou & Tsitsiklis, 1987), so there is little hope of finding a silver-bullet algorithm that will make things any easier.

One hope for handling POMDPs is the identification of special case algorithms that work with interesting subclasses of problems. HAMs can be applied directly to one special case, the case where *some* states can be identified with certainty. If a HAM is designed so that choices can be made only at these states, then the problem of optimally refining a HAM will remain an SMDP, since there will be no points in the reduced model for which the agent will never be uncertain about the state of environment. This dodges the problem of partial observability, but it may be applicable in cases where the environment contains distinctive landmarks.

Another route around the problem of partial observability is by augmenting the machines in a HAM to incorporate a limited memory of past experiences. Wiering and Schmidhuber (1996) propose a method for learning complex state machines for POMDPs through reinforcement learning. This method uses some methods similar to those used in HAMs, but its formal properties are unclear. Recent advances in the manipulation of POMDP policies as state machines (Hansen, 1997) may give further insight into this issue.

The algorithms for combining a HAM with an MDP to produce a reduced SMDP can be applied to combine a HAM with a general POMDP to produce a reduced Partially Observable Semi-Markov Decision Process (POSMDP), the solution of which optimally refines the HAM for the original POMDP. Since POMDP algorithms scale very badly with the number of states, this could be a useful approach. One difficulty can arise from the use of sensor information by HAMs to decide where to create choice points. HAMs reduce the state space by limiting the number of points at which choices are made. If the environment can generate confusing observations, then the HAM might wind up generating choice points for all states in the environment, eliminating any hopes of simplifying the problem. This problem may be avoidable using approximation methods that disregard low-probability, i.e. noisy, sensor information (Zhang & Liu, 1997).

The application of the decomposition methods of Chapter 6 to POMDPs would be very complicated. In general, there may exist strings of observations that could cause an agent's belief state to become smeared out over the entire state space. This would thwart efforts to treat different parts of the state space as independent problems. As with HAMs one workaround may be through some combination of decomposition methods and POMDP approximation methods.

7.2 Conclusion

The new technical content of this dissertation began with a discussion of the concept of temporal abstraction. Temporal abstraction was shown to be equivalent to the transformation of a policy defined over a region of the state space to an action in an SMDP. The existing methods for temporal abstraction were shown to be instances of this basic step.

Several algorithms were presented for efficiently computing the transformation required to convert a policy into an action. As a bonus, the investigation of the issues involved in this transformation suggested a new type of MDP optimality criterion that permits explicit tradeoffs between rewards and the discounted probability of policy outcomes.

The HAM method was introduced as a means of hierarchically generating temporally abstract actions. This method permits the partial specification of abstract actions in a way that corresponds to an abstract plan or strategy. Abstract actions specified as HAMs can be optimally refined for new tasks by solving a reduced SMDP. The implication of this transformation is that traditional MDP algorithms can be used to refine HAMs for new tasks in much less time than it would take to learn a new policy for the task from scratch.

While HAMs introduce hierarchical, temporally abstract actions, they do not decompose MDPs into *independent* subproblems. However, they do complement a new type of decomposition that can break apart MDPs into independent pieces. This decomposition method works by constructing a cache of policies for different regions of the MDP and then using the algorithms developed earlier in the dissertation to optimally combine the cached solution to produce a global solution.

Together, the methods developed in this dissertation provide important tools for producing good policies for large MDPs. Unlike many previous ad-hoc methods, these methods provide strong formal guarantees. They use prior knowledge in a principled way, and they reduce larger MDPs into smaller ones while maintaining a well-defined relationship between the smaller problem and the larger problem.

The author hopes that with these methods in hand, people who wish to apply MDP methods to practical problems can spend less time worrying about the vagaries of feature vectors for function approximators with unpredictable characteristics. Some of this effort can now be switched to thinking about how reusable components from previous problems can be applied to new problems and how problems can be described in ways that make them most amenable to decomposition. These efforts will payoff in high-quality solutions with meaningful performance guarantees.

Bibliography

Andre, D., Friedman, N., & Parr, R. (1997). Generalized prioritized sweeping. In *Advances in Neural Information Processing Systems 10: Proceedings of the 1997 Conference* Denver, Colorado. MIT Press.

Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 30–37 Tahoe City, CA. Morgan Kaufmann.

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1991). Real-time learning and control using asynchronous dynamic programming. Technical report TR-91-57, University of Massachusetts Computer Science Department, Amherst, Massachusetts.

Bellman, R. E. (1957). *Dynamic Programming.* Princeton University Press, Princeton, New Jersey.

Bertsekas, D. C., & Tsitsiklis, J. N. (1989). *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Englewood Cliffs, New Jersey.

Bertsekas, D. C., & Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming.* Athena Scientific, Belmont, Massachusetts.

Blackwell, D. (1962). Discrete dynamic programming. *Annals of Mathematical Statistics*, *33*, 719–726.

Boutilier, C., Dean, T., & Goldszmidt, M. (1995). Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pp. 1104–1111 Montreal, Canada. Morgan Kaufmann.

Boutilier, C., & Dearden, R. (1994). Using abstractions for decision-theoretic planning with time constraints. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)* Seattle, Washington. AAAI Press.

Boyan, J. A., & Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems 7: Proceedings of the 1994 Conference* Denver, Colorado. MIT Press.

Bradtke, S. J., & Duff, M. O. (1995). Reinforcement learning methods for continuous-time Markov decision problems. In *Advances in Neural Information Processing Systems 7: Proceedings of the 1994 Conference* Denver, Colorado. MIT Press.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, *2*, 14–23.

Cassandra, A. R., Kaelbling, L. P., & Littman, M. L. (1994). Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pp. 1023–1028 Seattle, Washington. AAAI Press.

Crites, R. H., & Barto, A. G. (1996). Improving elevator performance using reinforcement learning. In *Advances in Neural Information Processing Systems 8: Proceedings of the 1995 Conference* 1996. MIT Press.

Dantzig, G., & Wolfe, P. (1960). Decomposition principle for dynamic programs. *Operations Research*, *8(1)*, 101–111.

Dayan, P., & Hinton, G. E. (1993). Feudal reinforcement learning. In Hanson, S. J., Cowan, J. D., & Giles, C. L. (Eds.), *Neural Information Processing Systems 5*, pp. 361–368 San Mateo, California. Morgan Kaufman.

Dean, T., & Givan, R. (1997). Model minimization in Markov decision processes. In *Proceedings of the Fourteenth National Conference on Aritificial Intelligence*, pp. 106–111 Providence, Rhode Island. MIT Press.

Dean, T., Givan, R., & Leach, S. (1997). Model reduction techniques for computing approximately optimal solutions for Markov decision processes. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI-97)* Providence, Rhode Island. Morgan Kaufmann.

Dean, T., Kaelbling, L., Kirman, J., & Nicholson, A. (1995). Planning under time constraints in stochastic domains. *Artificial Intelligence*, *76*(1-2), 35–74.

Dean, T., & Lin, S.-H. (1995). Decomposition techniques for planning in stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pp. 1121–1127 Montreal, Canada. Morgan Kaufmann.

Dietterich, T. G. (1997). Hierarchical reinforcement learning with the MAXQ value function decomposition. Tech. rep., Department of Computer Science, Oregon State University, Corvallis, Oregon.

Forestier, J.-P., & Varaiya, P. (1978). Multilayer control of large Markov chains. *IEEE Transactions on Automatic Control*, *AC-23*, 298–304.

Gordon, G. J. (1995). Stable function approximation in dynamic programming. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 261–268 Tahoe City, CA. Morgan Kaufmann.

Hansen, E. (1997). An improved policy iteration algorithm for partially observable MDPs. In *Advances in Neural Information Processing Systems 10: Proceedings of the 1997 Conference* Denver, Colorado. MIT Press.

Harada, D. (1997). Reinforcement learning with time. In *Proceedings of the Fourteenth National Conference on Aritificial Intelligence*, pp. 577–582 Providence, Rhode Island. MIT Press.

Hauskrecht, M. (1998). Planning with temporally abstract actions. Tech. rep. CS-98-01, Computer Science Department, Brown University, Providence, Rhode Island.

Hauskrecht, M., Meuleau, N., Boutilier, C., Kaelbling, L. P., & Dean, T. (1998). Hierarchical solution of Markov decision processes using macro-actions. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98)*. To appear.

Jaakkola, T., Jordan, M., & Singh, S. P. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, *6*(6), 1185–1201.

Jaakola, T., Singh, S. P., & Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable Markov decision problems. In *Advances in Neural Information Processing Systems 7: Proceedings of the 1994 Conference* Denver, Colorado. MIT Press.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

Lin, L.-J. (1993). *Reinforcement Learning for Robots Using Neural Networks.* Ph.D. thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Lin, S.-H. (1997). *Exploiting Structure for Planning and Control.* Ph.D. thesis, Computer Science Department, Brown University, Providence, Rhode Island.

Lipton, R. J., Rose, D. J., & Tarjan, R. E. (1979). Generalized nested dissection. *SIAM Journal of Numerical Analysis*, *16*, 346–358.

Littman, M., Dean, T. L., & Kaelbling, L. P. (1995). On the complexity of solving Markov decision problems. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI-95)* Montreal, Canada. Morgan Kaufmann.

Littman, M. L. (1996). *Algorithms for Sequential Decision Making.* Ph.D. thesis, Computer Science Department, Brown University, Providence, Rhode Island.

Littman, M. L., Cassandra, A. R., & Kaelbling, L. P. (1996). Learning policies for partially observable environments: Scaling up. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 362–370 Tahoe City, CA. Morgan Kaufmann.

Lovejoy, W. S. (1991). A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, *28*(1–4), 47–66.

Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, pp. 159–196.

Mahadevan, S., & Connell, J. (1992). Automatic programming of behavior based robots using reinforcement learning. *Artificial Intelligence*, *55*(2-3), 311–365.

Mahadevan, S., Marchalleck, N., Das, T., & Abhijit, G. (1997). Self improving factory simulation using continuous-time reinforcement learning. In *Machine Learning: Proceedings of the Fourteenth International Conference*, pp. 202–210 Nashville, Tennessee. Morgan Kaufmann.

Marbach, P., & Tsitsiklis, J. (1998). Simulation-based optimization of Markov reward processes. Tech. rep., Laboratory of Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Moore, A. W., & Atkeson, C. G. (1993). Prioritized sweeping—reinforcement learning with less data and less time. *Machine Learning*, *13*, 103–130.

Nilsson, N. J. (1994). Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, *1*, 139–158.

Papadimitriou, C. H., & Tsitsiklis, J. N. (1987). The complexity of Markov decision processses. *Mathematics of Operations Research*, *12*(3), 441–450.

Parr, R. (1996). Policy based clustering for Markov decision processes. In *Proceedings of the AAAI 96 Fall Symposium on Learning Complex Behaviors.*

Precup, D., & Sutton, R. S. (1997). Multi-time models for temporally abstract planning. In *Advances in Neural Information Processing Systems 10: Proceedings of the 1997 Conference* Denver, Colorado. MIT Press.

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

Ramadge, P. J., & Wonham, W. M. (1989). On the supervisory control of discrete event systems. *Proceedings of the IEEE*, *77*(1), 81–98.

Russell, S. J., & Norvig, P. (1995). *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, New Jersey.

Singh, S. P. (1992). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, *8*(3), 323–340.

Singh, S. P., & Gullapalli, V. (1993). Asynchronous modified policy iteration with single-side updates. Unpublished manuscript.

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

Sutton, R. S. (1990). Integrated architectures for learning, planning and reacting based on approximating dynamic programming. In *Machine Learning: Proceedings of the Seventh International Conference* Austin, Texas. Morgan Kaufmann.

Sutton, R. S. (1995). Temporal abstraction in reinforcement learning. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 531–539 Tahoe City, CA. Morgan Kaufmann.

Sutton, R. S., Precup, D., & Singh, S. P. (1998). Between MDPs and semi-MDPs: Learning, planning, and representing knowledge at multiple temporal scales. In prep.

Szepesvári, C., & Littman, M. L. (1996). Generalized Markov decision processes: Dynamic programming and reinforcement-learning algorithms. Tech. rep., Computer Science Department, Brown University, Providence, Rhode Island.

Tate, A. (1977). Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77)*, pp. 888–893 Cambridge, Massachusetts. IJCAII.

Tesauro, G. (1989). Neurogammon wins computer olympiad. *Neural Computation*, *1*(3), 321–323.

Thrun, S., & Schwartz, A. (1995). Finding structure in reinforcement learning. In *Advances in Neural Information Processing Systems 7: Proceedings of the 1994 Conference* Denver, Colorado. MIT Press.

Watkins, C. J. (1989). *Models of Delayed Reinforcement Learning.* Ph.D. thesis, Psychology Department, Cambridge University, Cambridge, United Kingdom.

White, D. J. (1993). *Markov Decision Processes.* Wiley, New York.

Wiering, M., & Schmidhuber, J. (1996). HQ-learning: Discovering Markovian subgoals for non-Markovian reinforcement learning. Tech. rep., Istituo Dalle Molle di Studi sull'Intelligenza Artificiale, Lugano, Switzerland.

Williams, R. J., & Baird, L. C. I. (1993). Tight performance bounds on greedy policies based on imperfect value functions. Tech. rep. NU-CCS-93-14, College of Computer Science, Northeastern University, Boston, Massachusetts.

Zhang, N., & Liu, W. (1997). Region-based approximations for planning in stochastic domains. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence (UAI-97)* Providence, Rhode Island. Morgan Kaufmann.

Zhang, W., & Dietterich, T. (1995). High-performance job-shop scheduling with a time-delay TD(λ) network. In *Advances in Neural Information Processing Systems 7: Proceedings of the 1994 Conference* Denver, Colorado. MIT Press.