

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

FINITE-MEMORY CONTROL OF PARTIALLY OBSERVABLE SYSTEMS

A Dissertation Presented

by

ERIC A. HANSEN

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

DOCTOR OF PHILOSOPHY

Department of Computer Science

UMI Number: 9909170

**Copyright 1998 by
Hansen, Eric Anton**

All rights reserved.

**UMI Microform 9909170
Copyright 1998, by UMI Company. All rights reserved.**

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

© Copyright by Eric A. Hansen 1998

All Rights Reserved

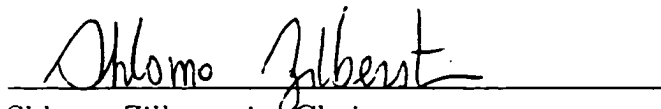
FINITE-MEMORY CONTROL OF PARTIALLY OBSERVABLE SYSTEMS

A Dissertation Presented

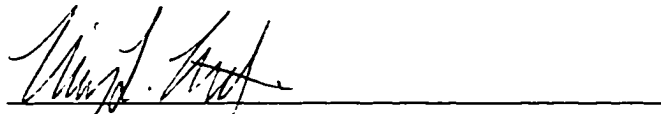
by

ERIC A. HANSEN

Approved as to style and content by:



Shlomo Zilberstein, Chair



Michael L. Littman, Member



Andrew G. Barto, Member



Victor R. Lesser, Member



Joseph Horowitz, Member



James F. Kurose, Department Chair
Department of Computer Science

ACKNOWLEDGMENTS

I would like to thank my advisor, Shlomo Zilberstein, for giving me the opportunity to complete my PhD. He supported my research for my last three years in graduate school and gave me the freedom to pursue my ideas. He provided feedback, wise advice and much encouragement. He continues to be an example of how to be a good professor and, more importantly, a good person.

I would also like to thank the members of my committee; Andrew Barto, Joseph Horowitz, Victor Lesser and Michael Littman. Michael, in particular, provided valuable help. His feedback on a paper that described an early version of the results of chapter 4 encouraged me to continue working on this topic. Over the past two years, he has generously shared his expertise and made several suggestions that influenced the direction of this work. I would also like to acknowledge Tony Cassandra for making available his value-iteration code and test examples.

I am fortunate to have a family that provided much moral support, and perspective, during my years in graduate school. I am grateful to them for that, and much more.

ABSTRACT

FINITE-MEMORY CONTROL OF PARTIALLY OBSERVABLE SYSTEMS

ERIC A. HANSEN

A.B., HARVARD UNIVERSITY

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Shlomo Zilberstein

A partially observable Markov decision process (POMDP) is a model of planning and control that enables reasoning about actions with stochastic effects and observations that provide imperfect information. It has applications in diverse fields that include artificial intelligence, operations research and optimal control, although computational difficulties have limited its use.

This thesis presents new dynamic-programming and heuristic-search algorithms for solving infinite-horizon POMDPs. These algorithms represent a plan or policy as a finite-state controller and exploit this representation to improve the efficiency of problem solving.

One contribution of this thesis is an improved policy-iteration algorithm that searches in a policy space of finite-state controllers. It is based on a new interpretation of the dynamic-programming operator for POMDPs as the transformation of a finite-

state controller into an improved finite-state controller. Empirically, it outperforms value iteration in solving infinite-horizon POMDPs.

Dynamic-programming algorithms such as policy iteration and value iteration compute an optimal policy for every possible starting state. An advantage of heuristic search is that it can focus computation on finding an optimal policy for a single starting state. However, it has not been used before to find solutions with loops, that is, solutions that take the form of finite-state controllers. A second contribution of this thesis is to show how to generalize heuristic search to find policies that take this more general form.

Three algorithms that use heuristic search to solve POMDPs are presented. Two solve special cases of the POMDP problem. The third solves the general POMDP problem by iteratively improving a finite-state controller in the same way as policy iteration, but focuses computation where it is most likely to improve the value of the controller for a starting state.

TABLE OF CONTENTS

	<u>Page</u>
ACKNOWLEDGMENTS	iv
ABSTRACT	v
LIST OF TABLES	x
LIST OF FIGURES	xii
 Chapter	
1. INTRODUCTION	1
1.1 Planning and feedback control	1
1.2 Thesis	4
1.3 Applications	6
1.4 Outline	8
2. MARKOV DECISION PROCESSES	11
2.1 The model	11
2.2 Policy representation	15
2.2.1 History policies	16
2.2.2 Bayesian policies	17
2.2.3 Finite-memory policies	18
2.3 Policy evaluation	24
2.4 Optimality equation	28
2.5 Computational complexity	28
3. DYNAMIC PROGRAMMING AND HEURISTIC SEARCH	31
3.1 Cyclic and acyclic sequential decision problems	31
3.2 Dynamic programming for completely observable MDPs	33

3.2.1	Value iteration	34
3.2.2	Policy iteration	36
3.2.3	Modified policy iteration	37
3.2.4	Action elimination	38
3.3	Algorithms for decision trees and acyclic decision graphs	39
3.3.1	Dynamic programming	42
3.3.2	Branch-and-bound	43
3.3.3	AO* – tree search	44
3.3.4	AO* – graph search	48
3.3.5	Receding-horizon control	49
3.4	Representation and reachability	50
4.	MULTISTEP DYNAMIC PROGRAMMING	53
4.1	Actions that reset memory	53
4.2	Multistep policy iteration	55
4.3	Performance	63
4.3.1	Machine maintenance	63
4.3.2	Gridworld	66
4.4	Discussion	72
5.	HEURISTIC SEARCH IN CYCLIC DECISION GRAPHS	74
5.1	An example the multistep approach cannot solve	74
5.2	LAO*	77
5.3	Related work: Completely observable MDPs	81
5.4	Discussion	83
6.	DYNAMIC PROGRAMMING FOR POMDPS	85
6.1	Value-function representation	86
6.2	Pruning	88
6.3	Dynamic-programming update	91
6.4	Value iteration	96
6.5	Sondik’s policy-iteration algorithm	97
7.	AN IMPROVED POLICY-ITERATION ALGORITHM	106
7.1	Policy iteration in a space of finite-state controllers	106
7.1.1	Value iteration and finite-state controllers	107

7.1.2	Policy improvement as change of a finite-state controller . . .	109
7.1.3	Example	112
7.1.4	Changing and merging memory states	113
7.1.5	Theoretical properties	114
7.1.6	Efficiency Issues	116
7.1.6.1	Policy-improvement preprocessing step	116
7.1.6.2	Policy evaluation: Decomposition of policy graph . .	117
7.1.6.3	Modified policy iteration	118
7.2	Performance	118
7.2.1	Marketing example	119
7.2.2	Machine-maintenance example revisited	122
7.2.3	4×3 gridworld	125
7.2.4	Approximate dynamic-programming updates	127
7.2.5	Tiger example revisited	129
7.2.6	Comparison to value iteration	132
7.3	Change of policy space	135
7.4	Question of finite convergence	137
8.	POLICY IMPROVEMENT USING HEURISTIC SEARCH . . .	140
8.1	Policy-improvement algorithm	142
8.1.1	The search algorithm	143
8.1.2	Improvement of the finite-state controller	144
8.1.3	Example	146
8.1.4	Theoretical properties	147
8.2	Performance	149
8.3	Discussion	153
9.	CONCLUSION	156
9.1	Summary of contributions	156
9.2	Future extensions	158
	APPENDIX: COMPUTATIONAL ENVIRONMENT	164
	BIBLIOGRAPHY	165

LIST OF TABLES

Table	Page
3.1 Value iteration for completely observable MDPs.	34
3.2 Policy iteration for completely observable MDPs.	36
3.3 AO* – tree search.	45
4.1 Multistep policy iteration.	57
4.2 Optimal multistep policy for gridworld of Figure 4.3.	67
4.3 Time to convergence in seconds as problem size increases.	69
4.4 Algorithm efficiency as cost for observation increases.	70
5.1 LAO*.	78
6.1 Linear program for testing vector dominance.	89
6.2 Algorithm for pruning a set of vectors.	90
6.3 Incremental pruning algorithm for dynamic-programming update. . . .	94
6.4 Algorithm for computing Bellman residual.	96
6.5 Sondik’s policy-iteration algorithm.	99
7.1 Improved policy-iteration algorithm.	110
7.2 Performance of policy iteration on Example 7.2.	126
7.3 Performance of policy iteration with relaxed precision parameter on Example 7.2.	128

7.4	Comparison of how soon policy iteration and value iteration converge to ϵ -optimality on eight small POMDPs.	133
8.1	Heuristic-search policy improvement.	142

LIST OF FIGURES

Figure	Page
2.1 Example of (a) memory tree, (b) policy and (c) corresponding finite-state controller.	19
2.2 Comparison of information-state controller and finite-state controller. .	22
2.3 Markov chain for a two-action and two-observation POMDP for which a policy visits two information states.	25
2.4 (a) Finite-state controller and (b) corresponding Markov chain for two-state, two-action and two observation POMDP.	26
2.5 Piecewise linear and convex value function.	27
3.1 Example decision tree for two-action and two-observation POMDP. . .	40
4.1 Optimal multistep policy for maintenance problem.	65
4.2 Multistep policy for maintenance problem with changed parameters. . .	66
4.3 Hundred-state gridworld with states numbered for reference to Table 4.2.	67
5.1 Optimal finite-state controller for Example 5.1.	75
6.1 Example of a piecewise linear and convex value function for a two-state POMDP.	86
6.2 Non-minimal representation of a piecewise linear and convex value function for a two-state POMDP.	88
7.1 Example of an acyclic finite-state controller for a finite-horizon POMDP.	108

7.2	Extraction of a cyclic finite-state controller after convergence of value iteration.	109
7.3	Example of policy improvement using dynamic-programming update.	112
7.4	Strongly-connected components in policy graph.	117
7.5	Parameters for Example 7.1.	119
7.6	Convergence of policy iteration on Example 7.1.	121
7.7	Convergence of policy iteration on Example 4.1.	123
7.8	Optimal policy for Example 4.1 with changed parameters.	125
7.9	Gridworld from Russell and Norvig [97].	125
7.10	Optimal policy for Example 5.1 found by policy iteration.	129
7.11	Convergence of error bounds (above) and number of vectors (below) for Example 5.1.	130
8.1	Example of policy improvement using heuristic search.	147
8.2	Convergence of bounds for 4×3 grid problem using heuristic search and policy iteration.	150
8.3	Convergence of bounds for network problem using heuristic search and policy iteration.	151
8.4	Convergence of bounds for network problem using two different upper-bound functions.	152

CHAPTER 1

INTRODUCTION

1.1 Planning and feedback control

Planning is the process of formulating a course of action to achieve an objective. It is a hallmark of intelligence and people do it constantly in their everyday lives. It is equally pervasive in engineering, industry, and management.

Automatic planning is the treatment of planning as a computational process. It involves the design and use of algorithms for constructing plans. Algorithmic construction of a plan requires a formal model of a planning problem that represents the relevant states, the possible courses of action, their likely outcomes, and the objectives, in a precise form that can be manipulated by a computer program. Many problems resist such formalization. But for problems that do not, automatic planning can be very useful.

For example, planning the operation of communications antennas for NASA's Deep Space Network or project management planning for spacecraft assembly, integration, and verification are problems with hundreds of variables that are inter-related in complex ways [1]. For such problems, a computer-generated plan can be superior to a human plan because a computer is better able to manage this kind of complexity. Planning is also a hallmark of autonomous behavior and that makes automatic planning an integral part of cutting-edge research on autonomous agents. It is essential for robots that move about on their own, through hallways or on the surface of Mars, as well as for software agents that ply the Internet [55, 115].

For these reasons and others, planning has been a central area of study in artificial intelligence (AI) from its inception. Early research adopted an approach that has come to be called “classical planning.” It is characterized by several simplifying assumptions: perfect knowledge, actions with deterministic effects, and objectives that can be modeled as simple goals to be achieved. Given these assumptions, a plan takes the form of a sequence of actions that leads deterministically from a start state to a goal state.

Over the past decade, the attention of AI researchers has turned to more complex planning problems that are characterized by uncertainty of various kinds and by the need to make tradeoffs between competing objectives. This has led to interest in decision-theoretic approaches to planning in which probability theory is used to represent uncertainty and utility theory is used to represent competing objectives [45, 32]. It has also led to a convergence between AI research on planning and related work in operations research, control theory, and the decision sciences, where the problem of sequential decision making under uncertainty has been studied for years.

An example of this convergence is the adoption of the *Markov decision process* (MDP) as a framework for decision-theoretic planning [21, 23, 50].¹ This model of decision making has been studied in operations research, optimal control theory, and related fields for many years [5, 46, 93, 6]. It models a controller (agent or decision maker) that interacts with a system (environment or plant) by taking a sequence of actions to optimize a performance objective.² The interaction takes the form of a feedback loop in which the controller observes the outcome of each action before

¹The MDP model has also been adopted in the AI community as a framework for reinforcement learning, a branch of machine learning that studies how agents learn to maximize reward from trial-and-error interaction with an environment [106].

²The terms “agent” and “environment” are common in artificial intelligence. The terms “controller” and “plant” are used in control theory, and the terms “decision maker” and “system” are used in operations research. I will use these terms interchangeably, with a preference for “controller” and “system.”

taking its next action. Feedback compensates for uncertainty about the effects of actions and about the state of the system.

A solution to a planning problem formalized as an MDP is a *policy* that specifies what action to take in response to feedback about the state of the system. In this thesis, I will use the terms *policy* and *plan* interchangeably to refer, as both do, to a representation of behavior that is intended to achieve an objective or goal. I will also use the terms *planning* and *control* interchangeably. Concern with the role of feedback in plan execution has stimulated growing recognition of the relationship between planning and control theory, in particular, between the problem of planning under uncertainty and the problem of stochastic optimal control [22]. Use of the MDP model as a framework for planning reflects this convergence and can be viewed as the adoption of closed-loop control for planning under uncertainty, as an alternative to classical planning which can be viewed as a form of open-loop control.

The standard MDP model assumes a particularly simple form of feedback. It assumes that feedback is received after each action and that it provides perfect and complete information about the current state of the system. An MDP that satisfies this assumption is said to be *completely observable*. There are problems for which this assumption is reasonable, just as there are problems for which the assumptions of classical planning are reasonable. But there are many problems for which both models are unrealistic. In contrasting ways, each makes overly-simplistic assumptions about the role of feedback in plan execution. The classical planning model assumes that feedback is unnecessary; the MDP model assumes it is perfect and complete. Both assumptions simplify planning by making it unnecessary to plan to acquire feedback.

A *partially observable Markov decision process* (POMDP) is a generalization of the standard MDP model that does not make these simplifying assumptions. It can be used to model planning and control problems for which feedback may provide imperfect or incomplete information about the state of a system. By modeling the

information-gathering effects of actions as well as their control effects, this model makes it possible to reason about the costs and benefits of acting to gain state information versus acting to change the system state. It also brings the process of information gathering directly under the control of the planner. For a completely observable MDP, feedback is perfect and passive. For a POMDP, feedback is imperfect and actively controlled by the planner. A POMDP provides a richer and more realistic model of the relationship between planning and information gathering, one that encompasses both closed-loop and open-loop control and applies to the wide range of problems for which neither the simplifying assumptions of classical planning nor the simplifying assumption of complete observability can be made.

Despite the expressiveness of the POMDP model, it has found limited use because of computational difficulties associated with it. Explicit modeling of both the information-gathering and control effects of actions dramatically increases the complexity of planning. At present, the best algorithms for solving POMDPs can only find optimal solutions for simple problems. Despite these computational difficulties, the POMDP model has been applied to several problems of practical significance. Some of these applications are reviewed in Section 1.3. If these computational difficulties could be overcome, it has many more potential applications.

1.2 Thesis

This thesis describes new dynamic-programming and heuristic-search algorithms for solving POMDPs. The unifying framework for these algorithms is the concept of finite-memory control. When an MDP is completely observable, an optimal policy takes the form of a simple, memoryless mapping from observations (which perfectly identify the current state) to actions. When an MDP is partially observable, the current observation does not provide sufficient information for optimal action selection. In this case, memory (or internal state) can improve decision making. A policy

conditioned on the internal state of a controller can be arbitrarily more complicated than a simple mapping from observations to actions. This makes POMDPs much more difficult to solve than completely observable MDPs.

The fact that memory can improve decision making for POMDPs is well-understood. The contribution of this thesis is to clarify the importance of *representation* of a plan or policy as a finite-state controller. In almost all previous work on planning and control of MDPs, a plan or policy is represented as a simple state-to-action mapping that does not show the structure of a plan. The claim of this thesis is that explicit representation of this structure, in the form of a finite-state controller, can be exploited to improve the efficiency with which POMDPs can be solved.

One advantage of this representation is that it makes policy evaluation for POMDPs straightforward. This provides the basis for an improved policy-iteration algorithm. A second advantage is that it makes analysis of reachability easier. Planning problems typically include a start state. Use of an admissible heuristic to prune regions of the state space that cannot be reached from the start state by following an optimal policy is a well-known technique of heuristic search. Three of the four algorithms described in this thesis use it to improve the efficiency with which POMDPs can be solved. Heuristic search has not been used before to find solutions with loops, that is, solutions that take the form of a finite-state controller. This generalization of heuristic search is an important contribution of this thesis.

Representation of a policy as a finite-state controller also makes it easier to relate work on solving MDPs to AI research on search and planning. It generalizes the classical representation of a plan as a sequence of actions leading from a start state to a goal state by adding branches, to express behavior that is conditional on the outcome of earlier parts of a plan, and loops, to express behavior that is repeated until some condition is met. This more general representation of a plan allows search for the best finite-state controller to be expressed as a graph-search problem. Thus, this

representation may help to bridge the gap between algorithms for solving POMDPs developed in the operations-research community and algorithms for planning and search developed in the AI community.

The focus of this thesis is on algorithms that use exact policy evaluation and converge, after a finite search or in the limit, to optimal solutions. This is a very high standard and unrealistic for most real-world applications. Only relatively simple problems can be solved by the algorithms presented in this thesis. However, these algorithms significantly outperform the best previous algorithms, and their success suggests that a similar approach to approximation is possible for larger and more complex problems. In particular, the success of the heuristic-search approach explored in this thesis suggests that more sophisticated AI planning techniques, among them abstraction and hierarchical organization, may be successfully applied within the framework established by this work.

1.3 Applications

A brief review of POMDP applications will help convey a sense of the scope of this model and the potential importance of finding improved algorithms.

The POMDP model was first studied in the operations-research community. Among the problems to which it has been applied, the most extensively studied are inspection and maintenance problems. The literature on such problems is vast and it is infeasible to cite more than a few references [112, 89, 96, 118]. However, the basic model for problems of this type is simple and can be explained briefly.

In an inspection/maintenance problem, a deteriorating system (for example, a piece of manufacturing equipment) is represented by a discrete Markov process where the states of the process represent degrees of deterioration. The simplest model has only two states: a new (or perfect) state and a failed state. More complex models allow intermediate states of deterioration. The system deteriorates stochastically in

the direction of the failure state and a decision maker has two different kinds of action to choose from to maintain it. One action corrects the deterioration of the system; it may be a *replace* action that restores the system to new or a *repair* action that possibly restores it to some intermediate state. The other action, usually called an *inspection*, provides perfect or imperfect information about the state of the system, usually at a cost. There are myriad variations of this model, but all have a form similar to this.

This simple model has many applications. The “deteriorating system” may represent a manufacturing machine or process where the quality of manufactured products depends on an internal machine state that can only be determined by halting manufacturing to perform an inspection [53, 72, 52, 118, 119]. It may represent a highway bridge where the safety of the bridge can only be imperfectly determined by visual or ultrasonic inspection [29, 49]. It may represent the progress of a disease where a costly medical test is needed to detect its presence or monitor its stage [60]. It may also represent the accounting status of a firm that can only be determined by a periodic audit [47]. The range of possible applications is one reason this simple model has been studied extensively. A second reason for interest in this model is that an optimal policy can be shown to have a simple “control limit” structure that may be exploited to find solutions more efficiently [96, 119, 70, 61].

Several other applications have motivated work on POMDPs. The POMDP model was first formulated for the problem of decoding signals received across a noisy communications channel [24]. Important early work was motivated by a simple teaching problem in which a student’s response provides imperfect information about what he or she has learned [101]. The POMDP model has also been applied to questionnaire design to deal with the possibility of untruthful answers [116]. Both of these applications illustrate the potential for POMDPs to model reasoning about hidden

mental state. Other applications include search for a moving target [27] and fisheries management [58].

While the foundations of the POMDP model were laid by the operations-research community, recent interest in this model in the AI community has brought a fresh perspective to the problem and a focus on its computational aspects. This has led to the development of improved dynamic-programming algorithms [14, 15, 35], machine-learning approaches to cope with model uncertainty [77, 80, 85, 100, 69], use of complexity analysis to characterize the difficulty of the problem [84, 64, 68], and various methods of approximation as an approach to solving larger and more complex problems [66, 38, 10]. Applications that have guided work on approximation are typically larger than those considered in operations research, and much larger than can be solved optimally. One application that has attracted considerable interest is mobile robot navigation. Because a robot's actuators have uncertain effects and its sensors provide partial and imperfect information, a POMDP provides a natural model for this kind of reasoning. Several working robots use the POMDP model in some form for control and navigation [55, 13, 44, 74]. The POMDP model has also been considered for applications such as automated driving [31, 80], medical therapy planning [40, 39, 114], and control of attention in machine vision systems [20].

1.4 Outline

The remainder of the thesis is organized as follows.

Chapter 2 introduces the MDP model and some of the concepts and notation used in this thesis. It focuses on the question of how to represent a policy and the possibility of representing it as a finite-state controller.

The rest of the thesis divides roughly into two parts. The first consists of chapters 3 through 5 and the second of chapters 6 through 8. Chapter 9 concludes the thesis.

Chapter 3 reviews dynamic-programming algorithms for completely observable MDPs and heuristic-search algorithms for decision trees and acyclic decision graphs. Chapters 4 and 5 describe two different ways of combining these algorithms to solve special cases of the POMDP problem.

In chapter 4, a generalization of dynamic programming for finite MDPs is described that uses heuristic search to perform backups. It is especially effective for an intermediate POMDP model in which observations provide perfect information but incur a cost and the problem is to determine when to make costly observations.

Chapter 5 describes a generalization of heuristic search that can find solutions with loops. There is a subset of POMDPs for which it is an effective and more general approach than the one developed in chapter 4, although it seems more promising as an approach to solving completely observable MDPs.

The second part of the thesis, which consists of chapters 6 through 8, describes a general approach to finite-memory control that is not limited to solving special cases of the POMDP problem.

Chapter 6 reviews dynamic-programming algorithms for POMDPs that rely on a piecewise linear and convex representation of the value function. These include a value-iteration algorithm for POMDPs that is widely used and a policy-iteration algorithm developed by Sondik [104, 103] that is not used because it is prohibitively complex and difficult to implement.

Chapter 7 takes a closer look at how to perform policy iteration for POMDPs. It shows that the difficulties of Sondik's policy-iteration algorithm hinge on its choice of policy representation and can be avoided by representing a policy as a finite-state controller. The policy-iteration algorithm it describes is simpler and more efficient than Sondik's algorithm. It also outperforms value iteration in solving infinite-horizon problems.

Chapter 8 describes an approach to POMDPs that combines the insights of the policy-iteration algorithm presented in chapter 7 with the heuristic-search approach developed in the first part of the thesis. The resulting algorithm can find a finite-state controller that optimizes the value of a starting state. This can be simpler and easier to find than a finite-state controller that optimizes the value of all possible possible starting states, which a dynamic-programming algorithm such as policy iteration tries to find.

Chapter 9 summarizes the contributions of the thesis, discusses possible extensions, and revisits the question of how AI research on planning and search is related to research on algorithms for solving POMDPs.

CHAPTER 2

MARKOV DECISION PROCESSES

This chapter introduces the Markov decision process model. It reviews the basic definitions and results of the model without reproducing theorems and proofs that can be found in standard references that explain this model more thoroughly [93, 6, 81, 71, 120]. The review focuses on the partially observable case and on two concepts that play a crucial role in this thesis: policy representation and policy evaluation. Algorithms for solving POMDPs are described in the chapters that follow.

2.1 The model

We consider a Markov decision process (MDP) consisting of the following elements:

- a finite set of states, S , of a system or process
- a finite set of actions, A , available to a controller or agent
- a *state transition function* that maps $S \times A$ into discrete probability distributions over S that represent uncertainty about the outcome of actions; let $Pr(s'|s, a)$ denote the probability that taking action $a \in A$ in state $s \in S$ results in a transition to state $s' \in S$
- a *reward function* that maps $S \times A$ into real numbers that represent expected rewards; let $r(s, a)$ denote the expected immediate reward for taking action $a \in A$ in state $s \in S$ (in some cases it is useful to define an equivalent cost function instead of a reward function)

- a start state denoted s_0

The state of a process is defined in such a way that it satisfies the *Markov assumption*. That is, the state contains all information about the history of the process that is relevant for predicting the resulting state after performing an action, as well as the immediate reward.

An MDP is said to be completely observable if an agent or decision maker always knows the state before taking an action. A partially observable MDP, or POMDP, is a generalization of this model that does not assume the agent knows the current state. Instead, it assumes the agent receives an observation that provides imperfect information about the state. A POMDP is created by augmenting the completely observable MDP model with the following elements:

- a finite set of observations, O
- an *observation function* that maps $S \times A$ into discrete probability distributions over O ; let $Pr(o|s', a)$ denote the probability that $o \in O$ is observed after taking action $a \in A$ resulting in a transition to state s'
- a discrete probability distribution, $\pi_0 = \{\pi_0(0), \pi_0(1), \dots, \pi_0(|S| - 1)\}$, over possible start states

A completely observable MDP can be viewed as a special case of a POMDP in which the state set and observation set are identical and the observation function is the identity function. In other words, it can be viewed as a POMDP in which each observation perfectly reveals the underlying state of the system.

This thesis considers the simplest POMDP model in which the set of states is finite, the set of actions is finite, the set of observations is finite, and the process unfolds at discrete time steps of equal duration. At the beginning of each time step t , the system is in some state $s_t \in S$. The decision maker takes action $a_t \in A$ and

receives a reward with expected value $r(s_t, a_t)$. The system then makes a transition to state s_{t+1} with probability $Pr(s_{t+1}|s_t, a_t)$ and the decision maker observes o_{t+1} with probability $Pr(o_{t+1}|s_{t+1}, a_t)$. The decision cycle repeats on the following time step.

It is possible to define MDPs that operate in continuous time or have infinite state, action, or observation sets [117, 18]. However, this introduces complications that are beyond the scope of this thesis. Although a POMDP can be solved by converting it into a special type of infinite-state completely observable MDP, as described in Section 2.2.2, the underlying system is still assumed to be in one of a finite number of states.

Performance criteria A *Markov decision problem* is a Markov decision process with an associated performance criterion. The performance criterion specifies how to combine rewards received over multiple time steps in order to evaluate performance. Together with the reward function, the performance criterion defines a utility (or value) function that expresses an agent's preferences, or equivalently, defines an objective function to be maximized.

Several performance 0 are possible. One that provides an important reference point in this thesis is expected total reward over a finite horizon H , defined as

$$E_{\pi_0} \left[\sum_{t=0}^H r(s_t, a_t) \right], \quad (2.1)$$

where E_{π_0} is the expectation operator, conditioned on the *a priori* state probability distribution. However, this performance criterion assumes the number of time steps for which the agent acts, the horizon H , is fixed and known in advance. For many planning problems, the objective is to act until a goal is achieved and the number of steps it takes to achieve the goal is not known in advance (and may be unbounded).

Such problems are better modeled using an infinite-horizon performance criterion and this thesis focuses on the most widely-used of these: expected discounted reward

over an infinite horizon. It is defined as:

$$E_{\pi_0} \left[\sum_{t=0}^{\infty} \beta^t r(s_t, a_t) \right], \quad (2.2)$$

where a discount factor $\beta \in [0, 1)$ ensures that the infinite sum of rewards is finite. The discount factor gives this performance criterion a mathematical property (called the “contraction property”) that makes it particularly convenient to analyze.

The idea of infinite-horizon performance includes the possibility that the problem never terminates, but does not require this. The idea of achieving an objective at some indeterminate time in the future, at which point the problem ends, is easily modeled using an infinite-horizon performance criterion. One way to do so is to add a special absorbing state to the state set that provides no reward no matter what the controller does. As soon as a controller achieves its objective, transition to this terminal state ensures that subsequent activity has no effect on the performance criterion and can be ignored. In this way, a special absorbing state can model termination of the problem.

Sometimes it is possible to show that a controller must eventually enter the terminal state. In that case, a discount factor is not needed to keep the total expected reward finite and it is possible to adopt expected total reward over an *indefinite horizon* as a performance criterion. Interestingly, discounting can be interpreted as a technique for ensuring that an agent eventually enters the terminal state. Discounting performance by some factor β is equivalent to assuming there is no discount factor, but an agent makes a transition to the terminal state with probability $1 - \beta$ each time step (and the remaining state transition probabilities are normalized).

The indefinite-horizon and the discounted infinite-horizon performance criteria are closely related, and the same algorithms can be used to solve both kinds of problem. This thesis focuses on these performance criteria. Another important infinite-horizon performance criterion, average reward per time step, requires a more complex al-

gorithmic approach. The possibility of extending the results of this thesis to the average-reward performance criterion is discussed briefly in the concluding chapter.

2.2 Policy representation

Solving a Markov decision problem means finding a rule for selecting actions that optimizes the performance criterion, or comes acceptably close to doing so. The rule for selecting actions is called a *policy* and is denoted δ .

For completely observable MDPs, a policy can take the simple form of a mapping from states to actions. This is possible because the state of the system is always known and satisfies the Markov property. When the performance criterion is to maximize indefinite-horizon or a discounted infinite-horizon return, there is an optimal policy that is deterministic (*i.e.*, the same state is always mapped to the same action) and stationary (*i.e.*, the mapping from states to actions is independent of time).¹ When the state set is finite, such a policy can be stored in a simple table that maps states to actions.

For POMDPs, a policy cannot take the form of a mapping from system states to actions because the system state is unknown or only partially observed. However a policy can still take the form of a mapping from states to actions if the state of the problem is defined differently. The word “state” is used in different senses in this thesis. In the first sense, it refers to the state of a partially observed system; the phrases “system state” and “state of the underlying system” convey this specific sense. In the second sense, it refers more broadly to a situation in which a decision must be made. There are three possible ways to define the state in this second sense: the entire history of the process, a probability distribution over possible system states,

¹For finite-horizon MDPs, an optimal policy is deterministic but usually non-stationary.

or the memory state of a finite-state controller. These three possibilities give rise to three possible representations of a policy for a POMDP.

2.2.1 History policies

The most general possibility is to make decisions based on the complete history of the process. If a controller knows the current system state, as it does for a completely observable MDP, knowledge of the previous history of the process cannot improve decision making and an optimal policy can be expressed as a simple mapping from system states to actions. But, if the system state is unknown or only partially observed, all of the previous history of the process may be relevant for decision making.

Let $h_t = \{\pi_0, a_0, o_1, \dots, a_{t-1}, o_t\}$ denote the history of the process up to time t , where π_0 is a prior probability distribution over possible starting states. Each time step, the history is updated as follows: $h_{t+1} = h_t \cup \{a_t, o_{t+1}\}$, with $h_0 = \{\pi_0\}$. Call h_t a *t-history*, let H_t denote the set of all t -histories, and let $H^* = \bigcup_{t=0}^{\infty} H_t$ denote the set of all possible histories. For a POMDP with a finite horizon H , a history-dependent policy δ is a sequence of functions $\{\delta_t\}_{t=0}^H$ such that for each $t = 0, 1, \dots, H$, $\delta_t : H_t \rightarrow \mathcal{A}$. For an infinite-horizon POMDP, a history-dependent policy $\delta : H^* \rightarrow \mathcal{A}$ maps the infinite set of all possible histories to the action set.

Redefining the state of the problem in this way creates a completely observable MDP with a state set that is the set of all possible histories. An optimal policy for this MDP is equivalent to an optimal policy for the original POMDP because it is based on all information available to the decision maker — the entire history of the process. However, defining the state of the problem in this way presents difficulties. For finite-horizon problems, the set of possible histories grows exponentially with the horizon, making analysis difficult. For infinite-horizon problems, the set of possible histories is infinite and it is not obvious how to represent a policy finitely in this case.

Before addressing the second difficulty, we review a solution to the first that involves summarizing the history of a process as a probability distribution over system states.

2.2.2 Bayesian policies

A probability distribution over system states, called a probability vector or *information state*, can be updated by Bayesian conditioning after each action and observation. Let π denote an information state and let $\pi(s)$ denote the probability that the current state of the system is s . If action a is taken in information state π and o is observed, each component of the successor information state π' is determined as follows,

$$\pi'(s') = \frac{Pr(o|s', a) \sum_{s \in S} Pr(s'|s, a) \pi(s)}{Pr(o|\pi, a)}, \quad (2.3)$$

where the denominator is a normalizing factor,

$$Pr(o|\pi, a) = \sum_{s' \in S} \left[Pr(o|s', a) \sum_{s \in S} Pr(s'|s, a) \pi(s) \right]. \quad (2.4)$$

As a convenient shorthand, let $T(\pi|a, o)$ denote this transformation of a prior probability vector π into a posterior probability vector π' given $a \in \mathcal{A}$ and $o \in \mathcal{O}$.

We discussed earlier the value of defining the state in such a way that it satisfies the Markov property, which means it conveys all information about the history of the process that is relevant for prediction and control. An information state that is updated by Bayesian conditioning satisfies the Markov assumption, that is, it is a *sufficient statistic* that summarizes all relevant information about the history of the process [105, 2, 103]. It follows that it is possible to define a completely observable MDP with a state set that consists of all possible information states, denoted $\Pi(S)$ or simply Π , an action set that remains \mathcal{A} , a transition function that is T , and a reward function that is defined for all information states as follows:

$$r(\pi, a) = \sum_{s \in S} \pi(s) r(s, a). \quad (2.5)$$

We call this more general MDP an *information-state MDP*. Because it is equivalent to the original POMDP, an optimal solution to the information-state MDP is an optimal solution to the original POMDP [105].

Summarizing relevant information about the history of a process as an information state makes analysis of a POMDP easier. Nevertheless, representing a policy as a mapping from the set of information states to the action set still presents a difficulty. The set of information states is continuous and uncountably infinite. A more convenient representation would map the set of all possible histories into a finite number of states.

2.2.3 Finite-memory policies

This thesis adopts a finite-state controller as a representation of a policy. A finite-state controller maps the set of all possible histories into a finite number of memory states. A memory state serves as a finite-valued statistic that summarizes the history of the process. Unlike an information state, it is not necessarily a sufficient statistic that contains all information about the history of the process that is relevant for decision making. But, the fact that the number of memory states is finite makes it easier to represent a policy in this way.

Finite-memory approaches to the POMDP problem have been investigated before and three types of finite-state controllers have been considered.

Memoryless policies The simplest approach to finite-memory control is not to use memory at all. A memoryless policy takes the form of a simple mapping from observations to actions. In this respect, it resembles a policy for a completely observable MDP which is likewise memoryless. The difference is that for POMDPs, observations do not reliably identify the current system state and do not satisfy the Markov assumption.

Although the minimalism of a memoryless approach to POMDPs is attractive, its effectiveness is limited. Littman [62] describes a combination of hill-climbing and branch-and-bound for finding deterministic memoryless policies that works well for several small examples. Singh, Jaakkola and Jordan [100, 48] describe a reinforcement learning algorithm that finds stochastic memoryless policies and show that a stochastic memoryless policy can outperform the best deterministic memoryless policy. However, both point out that the performance of a memoryless policy can be arbitrarily poor. For many POMDPs, some form of memory is needed for performance to reach an acceptable level.

Finite-length memory policies A second approach to finite-memory control is to base decision making on some finite string of the most recent actions and observations. Platzman [90] first explored this approach and describes an algorithm called “perceptive policy iteration” that he shows can solve the machine-maintenance example of Section 4.3.1. A similar policy representation is adopted by White and Scherer [121], who use value iteration to find a solution, and by McCallum [79], who uses reinforcement learning.

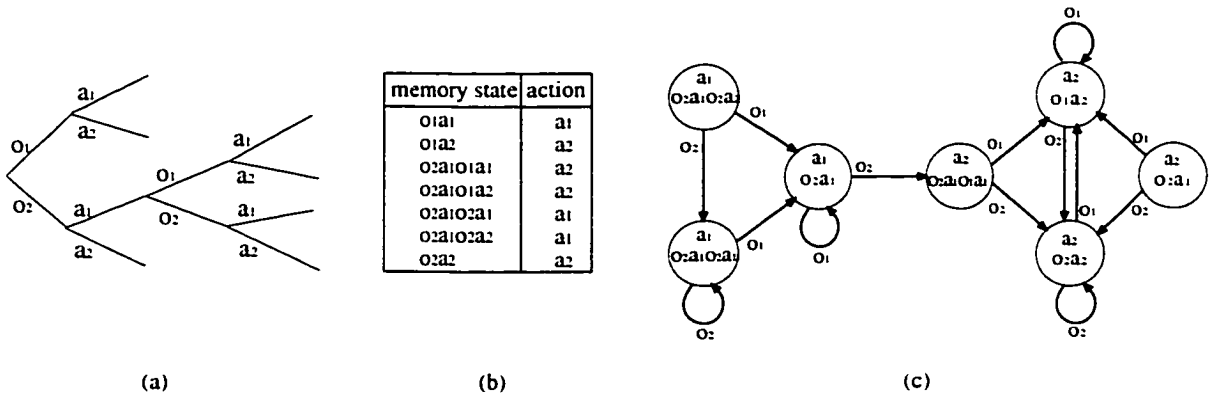


Figure 2.1. Example of (a) memory tree, (b) policy and (c) corresponding finite-state controller.

A finite-length memory policy can be represented by a *memory tree* in which each path from the root to a leaf represents a string of the most recent actions and obser-

uations. Figure 2.1(a) shows an example of a memory tree for a POMDP with two actions and two observations. The leaves of the tree need not have the same depth. Each leaf represents a memory state and growing a tree corresponds to adding new memory states by remembering longer sequences of actions and observations. A policy is created by mapping each memory state at a leaf of the tree to an action. Figure 2.1(b) shows a policy for this memory tree and Figure 2.1(c) shows the corresponding finite-state controller.

This is a more general representation than a memoryless policy (which corresponds to a memory tree of depth 1). However it is still a very limited representation. Not every finite-state controller can be represented by a memory tree. Each path from the root to a leaf of the tree represents a finite-length memory. But finite memory is a more general concept than finite-length memory. For example, the fact that a particular event occurred at some point arbitrarily long ago is easily stored in finite memory, but cannot be stored in finite-length memory. Thus, this approach to finite-memory control, although elegant, is also limited. Moreover, it impairs economy of representation. In order to remember something that occurred k steps ago, everything that happened in the intervening $k - 1$ steps must be remembered. This can give rise to a larger-than-necessary finite-state controller that makes combinatorially many irrelevant distinctions in order to make a single relevant distinction.

Finite-state controllers The most general approach to finite-memory control, and the one taken in this thesis, considers all possible finite-state controllers as potential policies. This approach has received less attention than the approaches already reviewed, but has not been entirely neglected.

Platzman [92] describes an approach to solving infinite-horizon POMDPs that uses linear programming to iteratively improve a policy represented by a stochastic finite-state controller. He reports solving a small POMDP with this approach. However, his algorithm is very complicated and I am unaware of any subsequent computational ex-

perience with it. Some attention has been given to using genetic algorithms [110] and related approaches [122] to evolve programs with internal state or memory, although the programs evolved do not take the specific form of a finite-state controller.

Both Sondik [104] and Cassandra *et al.* [14] note that dynamic-programming algorithms for POMDPs, which represent a policy as a mapping from information space to action space, sometimes converge to an optimal policy that is equivalent to a finite-state controller. These algorithms do not reliably converge to a policy that is equivalent to a finite-state controller, however, and it is not obvious how to use them to find an approximate or suboptimal finite-state controller. Nevertheless, the observation that dynamic programming sometimes converges to a policy that is equivalent to a finite-state controller is important because it hints that there is a relationship between dynamic-programming algorithms for POMDPs and representation of a policy as a finite-state controller.

A finite-state controller consists of the following elements.

- a finite set of inputs; we take these to be the set of observations, O , of the POMDP
- a finite set of outputs; we take these to be the set of actions, A , available to the controller
- a finite set of memory states, Q
- a memory state update function, $\tau : Q \times O \rightarrow Q$
- an output function, $\alpha : Q \rightarrow A$
- a nonempty set of possible starting memory states and a rule for selecting the starting memory state
- a possibly empty set of final memory states

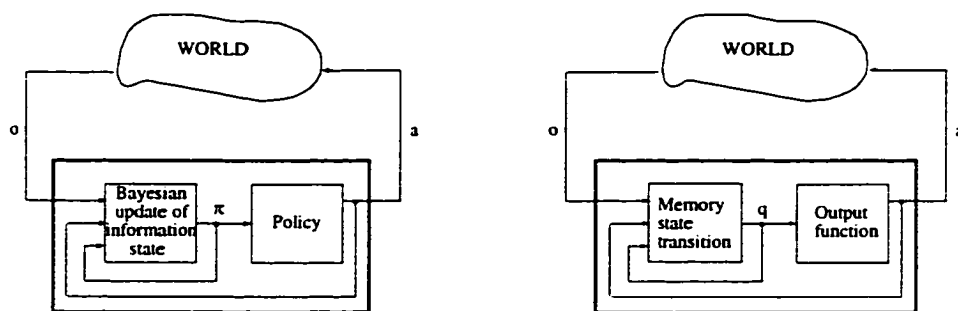


Figure 2.2. Comparison of information-state controller and finite-state controller.

Figure 2.2 reveals the similarity between an information-state controller and a finite-state controller for a POMDP. Because the output function maps memory states to actions, it plays a similar role as a policy that maps information states to actions. The memory-state-update function plays a similar role as the information-state-update function that uses Bayesian conditioning to map histories into information states. The difference is that the number of memory states is finite. The advantage is that a mapping from a finite set of memory states to actions is easier to represent than a mapping from a continuous information space to actions.

Representational simplicity is not the only advantage of a finite-memory approach. This approach also simplifies execution of a policy. Instead of having to maintain and update an information state at execution time, and use the information state to index a complex data structure that represents a policy defined for all of information space, execution of a finite-memory policy is straightforward. A finite-memory approach also makes policy evaluation easier and this last advantage is so important that we discuss it at length in Section 2.3. Before doing so, we point out there are also difficulties with this approach that have to be overcome.

Difficulties of the finite-memory approach There are two chief difficulties. The first is that a memory state is not necessarily a sufficient statistic. That is, it does not generally convey all information relevant for prediction and control of a POMDP.

For some problems, a finite-state controller can achieve optimal performance. But, in general, it cannot.

A couple of considerations mitigate this first difficulty. First, although a POMDP may not have an optimal finite-state controller, a finite-state controller can perform arbitrarily close to optimal by using arbitrarily many memory states. Second, a finite-memory approach makes possible a tradeoff between the number of memory states and the quality of decision making. Whereas an information state contains all information about the history of a process that is relevant for prediction and control, a finite-state controller can focus on the most relevant aspects of the history of a process and ignore those that are less important. Mapping different histories into the same memory state provides a form of generalization in which marginally relevant memory distinctions can be ignored. The fact that a memory state is not necessarily a sufficient statistic is a difficulty, but not necessarily a disadvantage; it offers intriguing possibilities for generalization and approximation.

A second, more compelling difficulty of the finite-memory approach is the difficulty of finding a good finite-memory representation for a problem. Finding a finite-memory policy does not simply mean finding a mapping from memory states to actions. It also means finding the memory-state-update function. There is only one way to perform Bayesian conditioning to update an information state. There is an infinite number of ways to classify the set of all possible histories into a finite number of memory states. Identifying the best classification function, that is, the best finite-memory representation for a problem, is the chief difficulty of this approach to solving POMDPs. It is the difficulty of determining how to organize limited memory and use it effectively in decision making, of deciding what to remember and what to forget, and the remainder of this thesis will address this problem.

Interacting finite automata It is worth briefly drawing attention to a particularly elegant aspect of a finite-memory approach to POMDPs. It represents the controller

and the system being controlled as interacting automata. Both a finite-state controller and an MDP consist of a finite number of states. Both receive an input each time step, make a state transition, and produce an output. The finite-state controller receives an observation as input, makes a transition to a new memory state, and produces an action as output. The system with which it interacts receives an action as input, makes a transition to a new state, and produces an observation as output.

There is an important difference between these two automata. The finite-state machine that represents the system or environment has a stochastic transition function and a stochastic output function; it is a stochastic automaton [88]. By contrast, the transition function and output function of the controllers considered in this thesis are deterministic. It is possible to define a finite-state controller that makes memory state transitions stochastically and chooses actions stochastically, and, for POMDPs, stochastic policies have some advantages [92]. In this thesis, I only consider deterministic policies. A possible extension of this work to stochastic policies is discussed in the concluding chapter.

2.3 Policy evaluation

The effect of a policy on the performance criterion is summarized by a *value function*. Value functions provide a way of comparing the performance of different policies and play a crucial role in all algorithms for solving MDPs. Let $V^\delta(\pi_0)$ denote the expected discounted reward over an infinite horizon for following policy δ starting from information state π_0 , defined as follows:

$$V^\delta(\pi_0) = E_{\pi_0} \left[\sum_{t=0}^{\infty} \beta^t r(\pi_t, \delta(\pi_t)) \right]. \quad (2.6)$$

Computing the value function for a policy is called *policy evaluation*.

For infinite-horizon MDPs that are completely observable, a policy determines a Markov chain in which each state of the chain corresponds to a state of the MDP.

The value of each state can be determined by solving the following system of $|S|$ linear equations in $|S|$ unknowns, in which there is one equation for each state of the Markov chain.

$$V^\delta(s) = r(s, \delta(s)) + \beta \sum_{s' \in S} Pr(s'|s, \delta(s)) V^\delta(s'), \forall s \in S. \quad (2.7)$$

These equations can be solved directly using a method such as Gaussian elimination or LU Decomposition, or they can be solved iteratively or in some other way.

For infinite-horizon POMDPs, the possibility of policy evaluation depends crucially on how a policy is represented. No general method is known for evaluating a policy that is represented as a mapping from information space to action space. But if a policy is represented as a finite-state controller, policy evaluation becomes straightforward. A finite-memory policy for a POMDP also determines a finite Markov chain and its value function can also be computed by solving a corresponding system of linear equations. This is perhaps the most important advantage of a finite-memory representation of a policy for POMDPs.

There are two possible ways that a policy for a POMDP can determine a finite Markov chain. They correspond to two approaches to finite-memory control of POMDPs developed in this thesis.

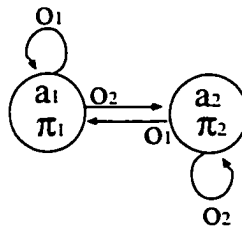


Figure 2.3. Markov chain for a two-action and two-observation POMDP for which a policy visits two information states.

The first possibility is that a policy started in a specific information state visits a finite number of other information states. Such a policy determines a finite Markov

chain in which each state of the Markov chain corresponds to a unique information state. The value of each information state is determined by solving a system of linear equations in which there is one equation for each information state visited by the policy. (See Figure 2.3.) This representation of the value function provides the foundation for algorithms for solving POMDPs described in chapters 4 and 5.

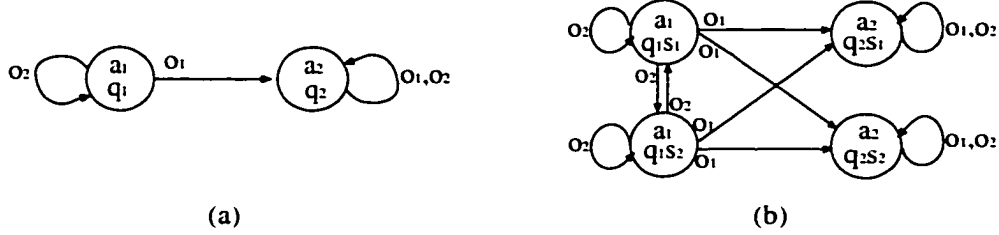


Figure 2.4. (a) Finite-state controller and (b) corresponding Markov chain for two-state, two-action and two observation POMDP.

The second possibility allows a finite-state controller to visit an infinite number of information states and supports a more general approach to solving POMDPs. In this method of policy evaluation, a finite-state controller determines a Markov chain in which each state of the chain corresponds to a combination of a memory state q_i and a system state s_j . Thus, the size of the Markov chain is $|Q||S|$. Figure 2.4 shows the structure of a Markov chain for a POMDP with two states, two actions and two observations and a finite-memory policy with two memory states. This method of policy evaluation computes a value function that is represented by a finite set of real-valued $|S|$ -dimensional vectors, denoted by Γ , where each vector $\gamma_i \in \Gamma$ corresponds to a memory state q_i of the finite-state controller. Thus the cardinality of Γ is $|Q|$. Policy evaluation consists of solving the following system of linear equations:

$$\gamma_i(s) = r(s, \alpha(i)) + \beta \sum_{s' \in S, o \in O} Pr(s'|s, \alpha(i)) Pr(o|s', \alpha(i)) \gamma_{\tau(i,o)}(s'), \quad (2.8)$$

where i is an index of a memory state of the finite-state controller, $\alpha(i)$ is the action associated with memory state i , and $\tau(i, o)$ is the index of the successor memory

state if o is observed. Figure 2.5 shows a value function with two vectors, one for each memory state.

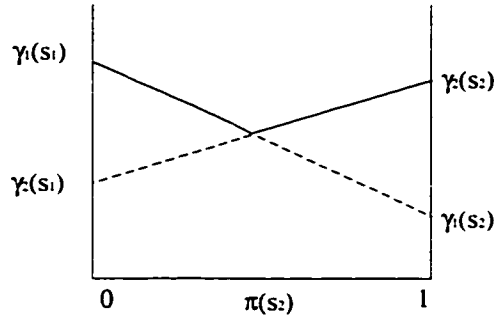


Figure 2.5. Piecewise linear and convex value function.

A value function computed in this way is defined for all of information space. For every possible starting information state π , the value function gives the value of starting the controller in memory state i , as follows:

$$V(\pi) = \sum_{s \in S} \pi(s) \gamma_i(s). \quad (2.9)$$

The value of starting the controller in each memory state is represented by a hyperplane in information space, making the value function piecewise linear. This reflects the fact that for any two information states, starting the controller in the same memory state results in expected values that differ by a linear function of the difference in the information states. If the finite-state controller is always started in a memory state that optimizes the value of the starting information state, the value function is convex as well as piecewise linear and the value of each possible starting information state is determined as follows:

$$V(\pi) = \max_{\gamma \in \Gamma} \sum_{s \in S} \pi(s) \gamma(s). \quad (2.10)$$

This method of policy evaluation is reviewed in more detail in chapter 6. It is the foundation for the algorithms for solving POMDPs described in chapters 6 through 8.

2.4 Optimality equation

There are many possible policies and the problem is to find the best one. Because the value functions of policies can be computed, we can compare policies to determine whether one is better than another. We also have a criterion for detecting an optimal policy. An optimal policy is defined as a policy that simultaneously optimizes the value of every information state. More formally, a policy δ^* is optimal if for every information state π and every policy δ , $V^{\delta^*}(\pi) \geq V^\delta(\pi)$. An optimal value function satisfies the *Bellman optimality equation*. For completely observable MDPs, the optimality equation takes the form,

$$V^*(s) = \max_{a \in A} \left[r(s, a) + \beta \sum_{s' \in S} Pr(s'|s, a) V^*(s') \right], \quad (2.11)$$

where $V^*(.)$ denotes the optimal value function. For POMDPs it takes the form,

$$V^*(\pi) = \max_{a \in A} \left[r(\pi, a) + \beta \sum_{o \in O} Pr(o|\pi, a) V^*(T(\pi|a, o)) \right]. \quad (2.12)$$

All of the dynamic-programming and heuristic-search algorithms described in this thesis use the Bellman optimality equation as a set of constraints on state values and solve an MDP by adjusting state values to satisfy these constraints.

2.5 Computational complexity

The completely observable MDP problem can be solved efficiently in the sense that its computational complexity is polynomial in the size of the state and action sets. (See [67] for a survey of complexity results for completely observable MDPs.)

The introduction of partial observability increases the complexity of the MDP problem dramatically. Papadimitriou and Tsitsiklis [84] show that the finite-horizon POMDP problem is PSPACE-complete and Littman [64] shows that the infinite-

horizon problem is EXPTIME-hard. The latter result is a lower bound on the complexity of the infinite-horizon problem and the upper bound is unclear. The problem may be undecidable although this has not yet been shown for discounted infinite-horizon POMDPs.

The possibility that the infinite-horizon problem is undecidable corresponds to the possibility that existence of an optimal finite-state controller cannot be decided for every instance of the problem. Even if an optimal finite-state controller exists for a particular problem instance, the complexity of finding it is closely related to the difficulty of bounding its size. That is, the time complexity and the memory complexity of the POMDP problem are closely related. This makes intuitive sense. POMDPs are much more difficult to solve than completely observable MDPs because a policy conditioned on internal state can be arbitrarily more complicated (as measured by the number of memory states) than a simple mapping from observations to actions.

Papadimitriou and Tsitsiklis [84] show that the size of the data structure needed to represent an optimal solution for a finite-horizon POMDP cannot be bounded by a polynomial in the size of the input. The same holds for the infinite-horizon case: an optimal finite-state controller, if one exists, may have an impractical number of memory states. This points to a more reasonable question: what is the complexity of finding the best policy that has a size bounded by a polynomial in the size of the input? For infinite-horizon POMDPs, the complexity of this revised question is PSPACE-complete instead of EXPTIME-hard, somewhat of an improvement. As a general strategy, Littman *et al.* [68] suggest that “there is some hope of building effective planning algorithms by ... searching for small controllers instead of optimal policies.”

These complexity results are worst-case results and not all POMDPs are equally difficult to solve. Nevertheless, these worst-case complexity results provide an important context for this thesis. The algorithms described in the following pages can find

policies that are optimal, or arbitrarily close to optimal, and can do so more efficiently than the best algorithms in current use. But they can only find optimal (or close-to-optimal) solutions for problems with simple solutions. Given these complexity results, that is not surprising.

Nevertheless, the relevance of the POMDP model for many practical problems means the most important question is not, how hard is it to find an optimal solution? It is, instead, how do we find the best solution possible given a bound on computation time and memory? In the concluding chapter, I discuss how the approach developed in this thesis may provide a framework for tackling this more reasonable question.

CHAPTER 3

DYNAMIC PROGRAMMING AND HEURISTIC SEARCH

The first part of this thesis describes an approach to finding finite-memory policies that works well for a small, but significant, subset of POMDPs. The approach is developed in chapters four and five and combines dynamic programming and heuristic search in a novel way. This chapter lays its groundwork by reviewing dynamic programming and heuristic search separately and discussing their relationship. The relationship between dynamic programming and heuristic search also figures prominently in the later chapters of this thesis.

3.1 Cyclic and acyclic sequential decision problems

Dynamic programming and heuristic search are two methods for solving sequential decision problems. Both solve a sequential decision problems by solving, in different ways, a system of nonlinear equations called the *Bellman optimality equation*. (See Section 2.4.) This optimality equation defines a set of constraints on optimal state values. Both dynamic programming and heuristic search adjust state values until they satisfy these constraints using an operation called a *backup*. A backup can be viewed as the transformation of an equation in the Bellman system into an assignment statement that evaluates the expression on the right-hand side and assigns its value to the state on the left-hand side. For example, the optimality equation for POMDPs with discounting is,

$$V^*(\pi) = \max_{a \in A} \left[r(\pi, a) + \beta \sum_{o \in O} Pr(o|\pi, a) V^*(T(\pi|a, o)) \right], \quad (3.1)$$

and a backup is the similar assignment statement:

$$V(\pi) := \max_{a \in A} \left[r(\pi, a) + \beta \sum_{o \in O} Pr(o|\pi, a) V(T(\pi|a, o)) \right]. \quad (3.2)$$

Because the states on the right-hand side of each equation are possible successors of the state on the left-hand side, a state value is said to be computed by “backing-up” the values of its successor states.

The successor relationship between states imposes an ordering on the state values that must be backed-up to solve the optimality equation. For some sequential decision problems, it is a partial ordering. For other problems, in particular, for infinite-horizon problems, it is not. The distinction is important because it influences how each type of sequential decision problem can be solved.

Both types of problem can be solved using dynamic programming. However problems for which the state values that must be backed-up to solve the optimality equation can be partially ordered — call them *acyclic sequential decision problems* — can be solved by a more straightforward and efficient dynamic-programming algorithm called *backwards induction*. Beginning with a set of terminal state values, all other state values must be backed-up just once to solve the optimality equation. Problems for which these state values cannot be partially ordered — call them *cyclic sequential decision problems* — must be solved using a more complex dynamic-programming approach called iterative relaxation that backs-up estimated state values repeatedly before eventual convergence.

In the case of heuristic search, the distinction between cyclic and acyclic sequential decision problem is even more critical. Heuristic-search algorithms have been developed for acyclic sequential decision problems only, not for cyclic problems. One of the contributions of this thesis is to generalize heuristic search to solve cyclic sequential decision problems.

This chapter reviews examples of each type of sequential decision problem. Section 3.2 reviews dynamic-programming algorithms for completely observable MDPs. This is an example of a cyclic sequential decision problem. Section 3.3 reviews algorithms for decision-tree problems. This is an example of an acyclic sequential decision problem. Both dynamic-programming and heuristic-search algorithms have been developed for decision-tree problems. Comparing them helps to clarify the similarities and differences between these two problem-solving techniques and provides guidance about how to combine them. The approach developed in chapters 4 and 5 combines the heuristic-search algorithms for finite decision trees reviewed in Section 3.3 with the dynamic-programming algorithms for infinite-horizon MDPs reviewed in Section 3.2.

3.2 Dynamic programming for completely observable MDPs

For completely observable MDPs with a finite state set and a discounted infinite-horizon performance criterion, the optimality equation is given by equation 2.11 and, for each state s , a backup takes the following form:

$$V'(s) := \max_{a \in A} \left[r(s, a) + \beta \sum_{s' \in S} Pr(s'|s, a) V(s') \right]. \quad (3.3)$$

Performing a backup for every state s in the state set is called a *dynamic-programming update*. It is the core step of dynamic-programming algorithms for solving MDPs. The fact that a backup is performed for every state in the state set is characteristic of dynamic programming. For infinite-horizon MDPs, the dynamic-programming update can be interpreted in two complementary ways — as the improvement of a value function or as the improvement of a policy. When interpreted as the improvement of a value function, it creates an algorithm called *value iteration*. When interpreted as the improvement of a policy, and when interleaved with policy evaluation, it creates an algorithm called *policy iteration*.

1. *Input:* An initial value function V^n with $n = 0$, and a parameter ϵ for detecting convergence to an ϵ -optimal value function.

2. *Improve value function:* Increment n and, for each $s \in S$, let

$$V^n(s) := \max_{a \in A} \left[r(s, a) + \beta \sum_{s' \in S} Pr(s'|s, a) V^{n-1}(s') \right].$$

3. *Convergence test:* If

$$\frac{2\beta \max_{s \in S} |V^n(s) - V^{n-1}(s)|}{1 - \beta} \leq \epsilon,$$

go to step 4. Otherwise, go to step 2.

4. *Output:* Extract an ϵ -optimal policy from the value function as follows. For each $s \in S$,

$$\delta(s) := \arg \max_{a \in A} \left[r(s, a) + \beta \sum_{s' \in S} Pr(s'|s, a) V^n(s') \right].$$

Table 3.1. Value iteration for completely observable MDPs.

3.2.1 Value iteration

Value iteration for infinite-horizon MDPs is best introduced by explaining value iteration for finite-horizon MDPs. For finite-horizon MDPs, an optimal policy and value function are non-stationary. Given a zero-horizon value function, V^0 , value iteration computes an n -horizon value function, V^n , for each horizon n up to some finite horizon H , using the dynamic-programming update to compute each V^n from V^{n-1} . Because state values are indexed by both the system state and the horizon, they are partially ordered. This makes the finite-horizon problem acyclic and each state value must be backed-up just once to find its optimal value.

For infinite-horizon MDPs, an optimal policy and value function are stationary. Instead of a separate value function and policy for each finite horizon n , the same value function and policy can be used every time step; essentially, the horizon is always the same — infinite. Value iteration for infinite-horizon MDPs can be viewed

as a generalization of value iteration for finite-horizon MDPs in the following way: as the horizon n recedes to infinity, the optimal n -horizon value function approaches the optimal stationary value function for the infinite-horizon problem. Value iteration for infinite-horizon MDPs is outlined in Table 3.1.

To guarantee convergence to optimal state values for an infinite-horizon problem, the dynamic-programming update must be iterated an infinite number of times. In practice, of course, value iteration must be stopped after a finite number of iterations. It can be shown that for any real number $\epsilon > 0$, value iteration converges to an ϵ -optimal value function after a finite number of iterations. A value function V^n is ϵ -optimal if

$$\max_{s \in S} |V^*(s) - V^n(s)| \leq \epsilon, \quad (3.4)$$

where $\max_{s \in S} |V^*(s) - V^n(s)|$ denotes the maximum difference, or error, between the optimal value function V^* and the n -step value function. Although it cannot be measured directly when the optimal value function is unknown, it can be estimated by measuring the so-called *Bellman residual*, defined as $\max_{s \in S} |V^n(s) - V^{n-1}(s)|$. It can be shown that if

$$\frac{2\beta \max_{s \in S} |V^n(s) - V^{n-1}(s)|}{1 - \beta} \leq \epsilon, \quad (3.5)$$

then $\max_{s \in S} |V^*(s) - V^n(s)| \leq \epsilon$. This provides a criterion for detecting the convergence of value iteration to ϵ -optimality.

Although value iteration computes a value function, a policy is easily extracted from it by adopting, for each state, the action a that maximizes its expected value based on one-step lookahead, as follows:

$$\delta(s) := \arg \max_{a \in A} \left[r(s, a) + \beta \sum_{s' \in S} Pr(s'|s, a) V^n(s') \right]. \quad (3.6)$$

1. *Input:* An initial policy δ .
2. *Evaluate policy:* Compute the value function V^δ for policy δ by solving the set of $|S|$ equations in $|S|$ unknowns given by equation 3.7.
3. *Improve policy:* For each state $s \in S$, if there is some action $a \in \mathcal{A}$ such that $[r(s, a) + \beta \sum_{s' \in S} Pr(s'|s, a)V^\delta(s')] > V^\delta(s)$ then $\delta'(s) = a$; otherwise, $\delta'(s) = \delta(s)$.
4. *Convergence test:* If δ' is the same as δ , go to step 5. Otherwise, set $\delta = \delta'$ and go to step 2.
5. *Output:* An optimal policy.

Table 3.2. Policy iteration for completely observable MDPs.

A policy created in this way is said to be a greedy policy with respect to the value function V^n . A greedy policy with respect to an optimal value function is optimal, and a greedy policy with respect to an ϵ -optimal value function is ϵ -optimal.

3.2.2 Policy iteration

Value iteration is sometimes called “search in value function space” because it iteratively improves a value function by reducing its difference from the optimal value function until a convergence criterion is satisfied; then it extracts a policy. Another algorithm, policy iteration, is sometimes called “search in policy space” because it iteratively improves a policy until a convergence criterion is satisfied.

Policy iteration requires three things that value iteration does not. First, it requires an explicit representation of a policy that is separate from representation of the value function. For completely observable MDPs with a finite state set, a policy $\delta : S \rightarrow \mathcal{A}$ can be represented by a simple lookup table. Second, it requires a method of policy evaluation. For completely observable MDPs, the value function, V^δ , of a stationary policy, δ , can be computed by solving the following system of $|S|$ linear equations in $|S|$ unknowns:

$$V^\delta(s) = r(s, \delta(s)) + \beta \sum_{s' \in S} Pr(s'|s, \delta(s)) V^\delta(s'). \quad (3.7)$$

Finally, policy iteration interprets the dynamic-programming update used by value iteration to improve a value function as the improvement of a policy. For completely observable MDPs, policy improvement consists of adopting for each state the action that optimizes its value in the dynamic-programming update.

Policy iteration for completely observable MDPs is summarized in Table 3.2. It interleaves the dynamic-programming update, used for policy improvement, with policy evaluation. Howard [46] introduced this algorithm and proved that whenever the current policy δ is not optimal, the dynamic-programming update finds a new policy δ' such that $V^{\delta'}(s) \geq V^\delta(s)$ for all states s and $V^{\delta'}(s) > V^\delta(s)$ for at least one state s . Because policy iteration improves the current policy each iteration and the number of possible policies is finite, it converges to an optimal policy after a finite number of iterations. It can be terminated before finding an optimal policy by testing for an ϵ -optimal policy. The test for ϵ -optimality is slightly different for policy iteration than for value iteration. For policy iteration, a policy δ' is ϵ -optimal if

$$\frac{\beta \max_{s \in S} |V^{\delta'}(s) - V^\delta(s)|}{1 - \beta} \leq \epsilon. \quad (3.8)$$

3.2.3 Modified policy iteration

Policy iteration usually converges in fewer iterations than value iteration. However, the policy-evaluation step can become a bottleneck as the size of the problem grows because its complexity is $O(|S|^3|A|)$, compared to the $O(|S|^2|A|)$ complexity of the dynamic-programming update. Because one step of policy iteration can take longer than one step of value iteration for problems with large state sets, value iteration can sometimes outperform policy iteration even when it takes more iterations to converge.

An algorithm called *modified policy iteration* combines the best features of value iteration and policy iteration. Instead of evaluating a policy by solving a system of linear equations exactly using an $O(|S|^3)$ method such as Gaussian elimination, it computes an approximate value function for a policy δ by updating each state value as follows:

$$V^\delta(s) := r(s, \delta(s)) + \beta \sum_{s' \in S} Pr(s'|s, \delta(s)) V(s). \quad (3.9)$$

Iterating this update of all state values some number k times makes the complexity of this iterative policy-evaluation step $O(k|S|^2)$.

Adjustment of the parameter k creates a spectrum of algorithms between value iteration and policy iteration. If $k = 0$, modified policy iteration is equivalent to value iteration. If $k = \infty$, an exact value function is computed and the algorithm is equivalent to policy iteration. Intermediate values of k can adjust a tradeoff between the time it takes to compute the value function of a policy more accurately and the value of doing so. If the tradeoff is adjusted well, modified policy iteration can outperform both value iteration and policy iteration.

3.2.4 Action elimination

We have seen that bounds on the optimal value function can be used to detect convergence to ϵ -optimality. The same bounds can also be used to accelerate computation of the dynamic-programming update.

The complexity of the dynamic-programming update is a function of the number of states and the number of actions that can be taken in each state. Action elimination is a technique for detecting that one or more actions cannot optimize the value of a particular state and can be removed from the set of actions evaluated for that state on subsequent iterations of the dynamic-programming update [73].

There are two kinds of action elimination. Temporary action elimination removes an action from consideration on the following iteration only. Permanent action elimi-

nation removes an action from consideration on all subsequent iterations. I will briefly review permanent action elimination.

Permanent action elimination requires lower and upper bounds on the optimal value function. Recall that the Bellman residual, $\max_{s \in S} |V^n(s) - V^{n-1}(s)|$, can be used to compute bounds on the optimal value function. We already used these bounds to detect ϵ -optimality. An upper bound on the optimal value of any state s is

$$V^U(s) = V^n(s) + \frac{\beta \max_{s \in S} |V^n(s) - V^{n-1}(s)|}{(1 - \beta)}. \quad (3.10)$$

For value iteration, a lower bound on the optimal value function for any state s is

$$V^L(s) = V^n(s) - \frac{\beta \max_{s \in S} |V^n(s) - V^{n-1}(s)|}{(1 - \beta)}. \quad (3.11)$$

For policy iteration, a lower bound on the optimal value function is simply the current value function V^n .

Given upper and lower bounds on the optimal value function, action a can be eliminated from further consideration for state s if

$$\left[r(s, a) + \beta \sum_{s' \in S} Pr(s'|s, a) V^U(s') \right] < V^L(s). \quad (3.12)$$

This form of pruning, in which an upper bound is compared to a lower bound, is sometimes called *pruning by bounds*. Heuristic-search algorithms use it to prune states as well as actions.

3.3 Algorithms for decision trees and acyclic decision graphs

A representation of sequential decision making under uncertainty that is widely used in decision analysis is a decision tree. Figure 3.1 shows a simple example. By convention, squares represent *decision nodes* at which an action is chosen. Circles

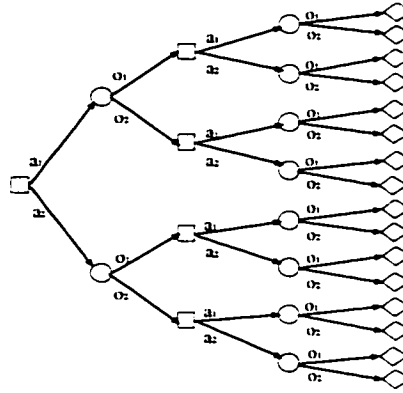


Figure 3.1. Example decision tree for two-action and two-observation POMDP.

represent *chance nodes* at which an observation is made. Diamonds represent *value nodes*; these are leaf nodes of the decision tree to which a priori values are attached.

A decision tree can represent a POMDP for which there is a starting information state and a set of terminal information states. Each node of the tree corresponds to an information state. The root corresponds to the starting information state and the other information states in the tree are determined by Bayesian conditioning. Values for terminal states are given with specification of the problem.

Because the nodes of the decision tree correspond to information states, the following notation will be useful. If an information state π is associated with a decision node, let π^a denote the information state associated with the successor chance node reached by taking action a . For this chance node, let π_o^a denote the information state associated with the successor decision (or value) node if observation o is made. Because each node corresponds to a unique information state, I will often refer to nodes as states.

The solution to a decision-tree problem also takes the form of a tree. It is often called a *solution tree*, or simply a plan, but I will call it a *policy tree* to emphasize the correspondence between a solution to a decision-tree problem and a policy for a MDP. (The phrase “policy tree” was introduced by Cassandra *et al.* [14] in a slightly different context.) A policy tree is defined as follows:

- the root node belongs to the policy tree
- if a decision node belongs to the policy tree, exactly one successor chance node belongs to the policy tree; it represents the choice of an action
- if a chance node belongs to the policy tree, all of its successor decision nodes belong to the policy tree; they represent the possible observations that can follow an action
- the leaves of the policy tree are leaves of the decision tree, that is, value nodes

There are combinatorially many possible policy trees in a decision tree. An optimal policy tree is defined by the following optimality equation:

$$f(\pi) = \begin{cases} V(\pi) & \text{for a terminal node} \\ \max_a [r(\pi, a) + \beta f(\pi^a)] & \text{for a decision node} \\ \sum_o Pr(o|\pi^a, a) f(\pi_o^a) & \text{for a chance node reached by taking action } a \end{cases}$$

Here, I use the notation $V(\pi)$ to denote the value of a terminal node of the decision tree and $f(\pi)$ to denote the value of any other node. If we consider a decision node and its child chance nodes as a unit that represents an action and the observations that can follow it, we can re-express the optimality equation as a lookahead of one time step.

$$f(\pi) = \begin{cases} V(\pi) & \text{for a terminal node} \\ \max_a [r(\pi, a) + \beta \sum_o Pr(o|\pi^a, a) f(\pi_o^a)] & \text{for a decision node} \end{cases}$$

A range of algorithms for finding an optimal policy tree are reviewed in the rest of this section. Some can find an optimal policy tree without generating the entire decision tree. This makes it useful to distinguish between the *implicit tree*, which is the entire decision tree that could be generated, and the *explicit tree*, which is that

portion of the decision tree that is actually generated by a search algorithm in the course of solving the problem. We use the term *tip node* to refer to a leaf of the explicit tree. We call a tip node that is also a leaf of the implicit tree a *terminal* tip node. Otherwise we call it a *nonterminal* tip node. A nonterminal tip node can be *expanded* to generate successor nodes that are added to the explicit tree. The entire tree, or any portion of it, can be generated by recursively expanding nonterminal tip nodes of the explicit tree.

3.3.1 Dynamic programming

In decision analysis, the familiar method for solving a decision-tree problem is called the *rollback method* and it is essentially dynamic programming. It first generates the entire decision tree (or assumes it has already been generated) and then, starting from terminal nodes for which values are given, solves the optimality equation by backing-up values recursively through the tree toward the root. A dynamic-programming algorithm that relies, like this, on a partial ordering of states to recursively back up state values is often called *backwards induction* and from now on I use this general term to refer to this algorithm.

Dynamic programming is said to be an *implicit-enumeration algorithm* because it finds an optimal policy without evaluating all possible policies, or in this case, all possible policy trees. It does so by using Bellman's *principle of optimality*. Once an optimal policy tree for a state has been found, this principle allows one to infer that an optimal policy that reaches this state must include the policy that is optimal for this state as a sub-policy. This is the rationale for solving the problem in a backwards order from the leaves of the tree to the root, recursively combining solutions to increasingly larger subproblems until the original problem is solved. Using Bellman's principle of optimality to avoid enumerating all possible policies is also called *pruning by dominance* and it is the hallmark of dynamic programming. Although the number

of possible policy trees grows exponentially with the number of nodes in a decision tree, the complexity of the dynamic-programming algorithm is linear in the number of nodes in the tree because each node is backed-up just once and each backup has constant-time complexity.

3.3.2 Branch-and-bound

Generating the entire decision tree before performing dynamic programming requires exponential memory in the depth of the tree. An optimal policy tree can be found in a more space-efficient manner by interleaving tree generation and evaluation. The decision tree can be generated in a depth-first manner and each node backed-up as soon as all its successor nodes have been backed-up, even if the entire decision tree has not yet been generated. Backing up nodes as soon as possible reduces the memory needed to store the decision tree because non-optimal branches can be pruned as soon as the optimal action for a decision node has been determined. A depth-first algorithm that interleaves decision-tree generation and evaluation, and prunes non-optimal subtrees based on the principle of optimality, uses memory proportional to the size of an optimal policy tree instead of memory proportional to the size of the entire decision tree. This space-efficient algorithm still generates and evaluates every node of the tree, however. This is characteristic of dynamic programming. Recall that the dynamic-programming algorithms for MDPs reviewed in Section 3.2 also evaluate every state. Pruning by dominance, upon which dynamic programming relies, is a technique for pruning policies, not states.

It is possible to prune states, and thus avoid generating the entire tree, by using a technique called *pruning by bounds*. Given an upper-bound function, the following rule can be used to prune a subtree before generating it: if an upper bound on the optimal value of one action branch is less than the backed-up value of another

action branch (which is a lower bound), the first action branch can be pruned without generating and evaluating its subtree.

Use of an upper-bound function for pruning gives rise to a depth-first branch-and-bound algorithm that usually does not need to evaluate the entire decision tree to find an optimal policy tree. We encountered the idea of pruning by bounds earlier in the form of action elimination. There it was used to prune actions only, not states that can be reached by taking actions. There are two reasons why we can now use it to prune states. First, we have assumed a starting information state and this changes the problem slightly; the problem now is to find an optimal policy for the starting state only, not for every possible state. Second, we have changed the representation of a policy from a simple mapping from states to actions to a tree. A tree representation of a policy shows which states are reachable from the starting state. Any state that cannot be reached from the starting state by following an optimal path through the tree can be pruned. This is our first example of how changing the representation of a policy can simplify problem solving.

3.3.3 AO* – tree search

There is a straightforward equivalence between the decision-tree representation of a sequential decision problem and the AND/OR tree representation developed in artificial intelligence. The equivalence is established by letting an OR node correspond to a decision node and an AND node correspond to a chance node. AND/OR trees were originally developed to represent and solve problems such as problem-reduction planning and theorem proving, but this equivalence makes it possible to use an AND/OR tree to represent a decision-tree problem as well. A best-first heuristic search algorithm for AND/OR trees, called AO*, can be used to find an optimal policy tree. (See Table 3.3.) Although the applicability of AO* to decision-tree problems often goes unrecognized, it has been exploited before [76, 60, 87, 94, 113].

1. *Input*: Starting information state.
2. *Initialization*: The explicit decision tree initially consists of a root node that corresponds to the starting information state.
3. *Forward search*. Expand the best partial policy tree as follows:
 - (a) *Identify best partial solution*. Identify the best partial policy tree by beginning at the root and always selecting the action branch with the greatest expected value (breaking ties in favor of the current policy).
 - (b) *Termination test*. If the best policy tree does not contain any nonterminal tip nodes, go to step 6.
 - (c) *Expand partial solution*. Select some nonterminal tip node of the best partial policy tree to expand and add its outgoing arcs and successor nodes to the explicit decision tree.
4. *Dynamic programming*. Create a set Z that contains the expanded node and all its ancestors in the explicit tree. Repeat the following until Z is empty.
 - (a) Remove a node from Z that has no descendent in Z and let π denote its corresponding information state.
 - (b) If a terminal node, set $f(\pi) := V^*(\pi)$.
 - (c) If a nonterminal tip node, set $f(\pi) := h(\pi)$.
 - (d) If a non-tip decision node, set $f(\pi) := \max_a [r(\pi, a) + \beta f(\pi^a)]$.
 - (e) If a non-tip chance node denoted π^a , set $f(\pi^a) := \sum_o Pr(o|\pi^a, a) f(\pi_o^a)$.
5. Go to step 3.
6. *Output*: An optimal policy tree.

Table 3.3. AO* – tree search.

Like branch-and-bound, best-first heuristic search can find an optimal policy tree without evaluating the entire decision tree. But it uses an upper-bound function for more than pruning. It also uses it to determine which tip node of the explicit tree to expand next. Instead of expanding nodes in a depth-first order, the idea of best-first search is to expand nodes in an order that maximizes the effect of pruning; the objective is to find an optimal policy while evaluating as little of the decision tree as possible. A “best-first” order for expanding the explicit tree is to expand the node most likely to be part of an optimal policy tree. This is made precise by defining the concept of a partial policy tree.

A *partial policy tree* of an AND/OR tree is defined as follows.

- the root node belongs to a partial policy tree
- if a decision node belongs to a partial policy tree, exactly one successor chance node belongs to the partial policy tree; it represents the choice of an action
- if a chance node belongs to a partial policy tree, all of its successor decision nodes belongs to the partial policy tree; they represent the possible observations that can follow an action
- the leaves of the partial policy tree are tip nodes of the explicit tree

This definition is similar to the definition of a policy tree, with the difference that the tip nodes of a partial policy tree may be nonterminal nodes of the implicit decision tree.

As with policy trees, there are many possible partial policy trees and a value function can be used to rank them. The value of a partial policy tree is defined similarly to the value of a policy tree. The difference is that if the tip node of a partial policy tree is nonterminal, it does not have a terminal value that can be backed-up. Instead, the upper-bound function is used to assign a value to the tip node and, in

this context, it is referred to as an *heuristic evaluation function*. By convention, this heuristic function is denoted h . It is said to be *admissible* if $h(\pi) \geq f^*(\pi)$ for all π . A heuristic estimate of the value of the best partial policy tree can be recursively calculated as follows:

$$f(\pi) = \begin{cases} V(\pi) & \text{for a tip node that is terminal} \\ h(\pi) & \text{for a tip node that is nonterminal} \\ \max_a [r(\pi, a) + \beta f(\pi^a)] & \text{for a non-tip OR node} \\ \sum_o Pr(o|\pi^a, a) f(\pi_o^a) & \text{for a non-tip AND node following action } a \end{cases}$$

By considering an action and the observations that can follow it a unit, we can rewrite this as:

$$f(\pi) = \begin{cases} V(\pi) & \text{for a tip node that is terminal} \\ h(\pi) & \text{for a tip node that is nonterminal} \\ \max_a [r(\pi, a) + \beta \sum_o Pr(o|\pi^a, a) f(\pi_o^a)] & \text{for a non-tip OR node} \end{cases}$$

Although an optimal policy tree and its value are not determined until the algorithm terminates, the best partial policy tree can be determined at any time by backing up heuristic estimates from the tip nodes of the explicit decision tree to the root. A best-first search algorithm always expands some tip node of the best policy tree. Because expanding a partial policy tree can decrease its value as new tip nodes with new heuristic estimates are added, AO* consists of two steps. First, it expands some node of the best partial policy tree. Then, it propagates the heuristic estimates of the new tip nodes back through the decision tree, possibly changing the best partial policy tree. The second step is the dynamic-programming algorithm already described in Section 3.3.1, except it does not evaluate all states — it just re-evaluates the ancestors of new tip nodes.

Efficiency considerations A few additional issues must be considered to implement AO* efficiently. First, a node can be marked *solved* when an optimal policy tree has been found for it. Marking a node as *solved* makes it possible to avoid unnecessary computation in the forward search step of AO* because there is no reason to search below a solved node for nonterminal tip nodes of the best partial policy tree. Pseudo-code for AO* usually includes a solve-labeling procedure, but I have left it out to clarify the essential algorithm.

Second, the best partial policy tree may have many nonterminal tip nodes. AO* works correctly no matter which of these tip nodes is chosen for expansion. However, the efficiency of AO* can be improved by using a good selection function to choose which nonterminal tip node to expand next. Some possibilities are:

1. select the tip node with the greatest upper bound
2. select the tip node with greatest probability of being reached
3. select the tip node for which the product of upper bound and probability of being reached is greatest

This list of possible node-selection functions is not exhaustive.

The memory requirements of AO* may grow exponentially with the size of the decision tree and memory-bounded versions of AO* have been described that avoid this problem [16]. It is also possible to use a weighted heuristic with AO* to adjust a tradeoff between solution quality and search time [17].

3.3.4 AO* – graph search

The version of AO* described in the previous section was developed for searching AND/OR trees and it finds a solution in the form of a tree [82]. A generalization of this algorithm for searching acyclic AND/OR graphs (or acyclic decision graphs) can find a solution in the form of an acyclic graph [76, 83]. I will call this solution

an acyclic policy graph, or just a policy graph, again adopting a term introduced by Cassandra *et al.* [14] in another context. In an acyclic graph, the same state can be reached by different paths from the root through the tree. Because any acyclic graph can be unfolded into an equivalent tree, both the tree-search and the graph-search versions of AO* solve the same class of problems. The relative advantage of graph-search over tree-search is that it avoids re-computing solutions when the same state can be reached along different paths through the tree.

Checking for subproblems that have been already solved (or in this case, for information states that have already been visited) incurs an overhead. Therefore, the graph-search version of AO* is not necessarily more efficient than the tree-search version. It is more efficient if there is some likelihood of encountering duplicate subproblems along different paths through the tree. How likely this is depends on the particular problem. For example, diagnosis/classification problems often require performing a sequence of tests where each test incurs a cost and the problem is to determine an optimal order for performing the tests. If the only effect of the tests is to provide information, the same set of test results gives rise to the same information state, regardless of the order in which the tests are performed. Therefore, many different paths through the decision tree can lead to the same information state. Martelli and Montanari [75] developed the first graph-search version of AO* for just this problem and Pattipati and Alexandridis [87] use a graph-search version of AO* to solve a similar problem.

3.3.5 Receding-horizon control

If we tried to represent an infinite-horizon POMDP as a decision tree, the tree would have infinite depth. The decision-tree algorithms reviewed in this section are for finite decision trees. Is there some way to extend the decision-tree approach to infinite-horizon problems?

One possibility is to generate a decision tree to some fixed depth, evaluate it, and make a decision for the information state at the root of the tree. After taking an action and observing its outcome, the process can be repeated by generating another finite decision tree for the resulting information state. In control theory, this strategy is referred to as *receding-horizon control*. In AI, it is commonly used for searching game trees and Korf's real-time heuristic search algorithm adapts it to single-agent search [56].

With this strategy, it is no longer possible to compute the exact value of a decision by solving the problem completely and there are no longer exact terminal values to back up. However, it is possible to compute upper and lower bounds on the value of a decision by backing up both upper and lower bounds from the leaves of the tree. Branches of the decision tree can be pruned whenever a lower bound on the value of a node exceeds an upper bound. Satia and Lave [98] first proposed this approach to infinite-horizon POMDPs and describe a branch-and-bound algorithm for receding-horizon control that can make ϵ -optimal decisions. Larsen [60] followed up on this work by examining the use of AO* for the same problem and Washington [113] uses AO* in a similar way.

This approach to decision making, in which search is interleaved with control by computing a single decision at a time, can be very effective. However, it is not pursued in this thesis. Instead, I focus on developing off-line algorithms that compute a policy or plan all at once. But in most cases, the same search algorithms can be used for both problems.

3.4 Representation and reachability

It is clear from this review of decision-tree algorithms that heuristic search has a big advantage over dynamic programming. Whereas dynamic programming must evaluate the entire decision tree to find an optimal policy, heuristic search can find an

optimal policy by evaluating only part of the tree. It does so by limiting the problem in two ways. First, it finds an optimal policy for a specific starting state rather than for all possible starting states. Second, it uses a heuristic to prune subtrees (regions of the state space) that are not reachable from the starting state when an optimal policy is followed. Expanding the decision tree in best-first order takes fullest possible advantage of the effect of pruning.

However, heuristic-search algorithms have not been developed for cyclic sequential decision problems such as infinite-horizon MDPs. For infinite-horizon MDPs, the closest thing to heuristic search is action elimination. It also uses an upper-bound function for pruning. But it prunes actions only, not states that can be reached by taking sequences of actions. I hinted earlier at the reason for this; its representation of a policy — a simple mapping from states to actions — is too impoverished to allow states to be pruned based on reachability.

Unlike dynamic-programming algorithms for finite MDPs, heuristic search algorithms represent a solution in a form that exhibits how states can be reached from each other. Representing a solution as a path or tree makes it possible to recognize, and prune, states that cannot be reached from the starting state by following an optimal policy. In the next two chapters, this technique for pruning unreachable states is extended to infinite-horizon problems. For this extension, our representation of a policy must be changed from a simple mapping from states to actions to something closer to the representations used by heuristic search. The policy tree (or acyclic policy graph) representation used by AO* can be generalized for infinite-horizon problems by allowing it to contain loops.

A policy that contains both branches and loops can also be viewed as a finite-state controller. If each memory state of the controller corresponds to a unique information state, the controller can represent an infinite-horizon policy that visits a finite number of information states. There is an interesting subset of POMDPs for

which an optimal infinite-horizon policy visits only a finite number of information states. For this subset of POMDPs, chapters 4 and 5 describe a synthesis of the heuristic-search and dynamic-programming algorithms reviewed in this chapter that can find optimal finite-memory policies.

CHAPTER 4

MULTISTEP DYNAMIC PROGRAMMING

This chapter and the next describe how to solve a subset of POMDPs by combining a dynamic-programming algorithm for infinite-horizon MDPs with a heuristic-search algorithm for decision-tree problems. The finite-memory policies found using this approach are limited in that each memory state corresponds to a unique information state. Therefore, only policies that visit a finite number of information states can be found with this approach. Nevertheless, there is an interesting subset of POMDPs for which an optimal policy takes this form.

4.1 Actions that reset memory

In general, the posterior information state of an action is determined by three things: the prior information state, the action, and the observation that follows. POMDPs that can be solved by the approach described in this chapter (and the next) have an action set that includes one or more actions with the following property: the posterior information state depends only on the action and the observation that follows. It does not depend on the prior information state and, therefore, it does not depend on the previous history of actions and observations. Such actions have the effect of “resetting” memory in the sense that a controller does not need to remember what happened before taking the action. If a policy requires taking an action that resets memory at finite intervals, it is clearly a finite-memory policy.

There are two different ways in which an action can have the effect of resetting memory. One is by virtue of its transition probabilities. The other is by virtue of its observation probabilities.

An action resets memory by virtue of its transition probabilities if it deterministically moves a system to a specific state, regardless of the state it is currently in, or if it deterministically causes a transition to a specific information state (representing uncertainty about the system state), regardless of the information state when the action is taken.

An action resets memory by virtue of its observation probabilities if the observation that follows it provides perfect information. This resets memory because identifying the current system state with certainty makes it unnecessary to remember what happened before the observation in order to improve state estimation.

Many POMDPs of practical significance include an action that resets memory in one or both of these ways. In particular, machine maintenance, quality control, and similar problems — a widely-studied class of problems in the operations-research literature — often (though not always) assume an inspection action that provides perfect information and/or a replace action that resets a process to a known state. (Monahan [81] reviews some of these models and their applications. Thomas *et al.* [112] reviews inspection models, including those that assume perfect inspections.)

This chapter describes an algorithm that can solve any POMDP for which an optimal policy resets memory at finite intervals. This is a sufficient condition, though not a necessary condition, for a POMDP to have an optimal finite-memory policy. Later chapters describe how to find optimal finite-memory policies for more general classes of POMDPs.

In most cases, an action that resets memory does so by providing perfect information about the system state, either because it resets the system to a particular state or because it is followed by an observation that provides perfect information.

This is not always the case, but I will assume this in the rest of the chapter because it simplifies notation at a small cost in generality. From now on, the reader should bear in mind that whenever I speak of an “action that provides perfect information,” the results can be generalized by substituting the phrase “action that resets memory” and adopting a more general notation.

4.2 Multistep policy iteration

This chapter describes an algorithm that can find policies that specify, for each system state, a policy tree such that every leaf of the policy tree is also a system state. In other words, the last action in each branch of the policy tree provides perfect information. Taken together, this finite set of policy trees, one for each system state, represents a finite-state controller.¹ Figures 4.1 and 4.2 near the end of this chapter show two examples.

A finite-memory policy that takes this specific form can be found efficiently using a value function that is defined for system states only, and by performing a dynamic-programming update that backs up the values of system states only, just as in the completely observable case. However, a traditional, single-step backup is not well-defined because the value function is not defined for the successor states of actions that do not provide perfect information. To deal with this, the concept of a *multistep backup* is introduced.

A multistep backup generalizes the familiar single-step dynamic-programming backup in the following way. Instead of finding the best action for a state using one-step lookahead, it finds the best policy tree for a state using multistep lookahead and backs up its value. With the restriction that the last action in each branch of

¹If a policy includes an action that resets memory to a state of imperfect information, it can be represented by specifying a policy tree for this state of imperfect information as well as for each system state.

the policy tree provides perfect information, a multistep backup is well-defined even if the value function is defined only for system states. By letting the system states serve as the terminal states of a decision-tree problem, a multistep backup can be performed efficiently using the decision-tree algorithms reviewed in Section 3.3.

For infinite-horizon problems, the number of possible policy trees is infinite unless some bound is placed on the depth of a policy tree. Therefore, a finite bound on the depth of lookahead is assumed. (Note that bounding the depth of a multistep lookahead is equivalent to bounding the interval between one action that provides perfect information and the next.) The bound can be adjusted between iterations to accelerate improvement and/or ensure convergence to an optimal policy. This is discussed at the end of this section.

I call a dynamic-programming algorithm that uses multistep backups *multistep dynamic programming* and I call a policy found using this approach a *multistep policy*. Both multistep value-iteration and policy-iteration algorithms are possible. However, policy iteration is more interesting because it relies on explicit representation of a policy as a finite-state controller and I will describe it in this chapter.

Table 4.1 outlines multistep policy iteration. It is simply policy iteration using multistep backups instead of single-step backups. In the rest of this section, some of the issues that must be considered to implement it efficiently are discussed.

Policy evaluation

A multistep policy can be evaluated by solving one of two possible systems of linear equations. In the more straightforward system of equations, there is an equation for each information state visited by the policy. (Because a multistep policy resets memory at finite intervals to one of a finite number of states, it can only visit a finite number of information states.) The value of each information state is defined by the

<ol style="list-style-type: none"> 1. <i>Input:</i> An initial multistep policy δ. 2. <i>Evaluate policy:</i> Compute the value function V^δ for policy δ by solving the set of S equations in S unknowns given by equation 4.1. 3. <i>Improve policy:</i> For each state $s \in S$, perform multistep backup with depth bound k. If an improved policy tree is found, change $\delta'(s)$ to the new policy tree. 4. <i>Convergence test:</i> <ol style="list-style-type: none"> (a) If δ' is the same as δ and the depth bound k has not been used to prune any search path, go to step 5. (b) Else, if δ' is the same as δ and the depth bound has been used to prune some search path, increase the depth bound and go to step 3. (c) Otherwise, set $\delta = \delta'$ and go to step 2. 5. <i>Output:</i> An optimal multistep policy.
--

Table 4.1. Multistep policy iteration.

expected reward for an action and the values of its successor information states, as described in Section 2.3.

In the second system of linear equations, there is an equation for each system state. Each equation defines the value of a system state in terms of the expected reward for the policy tree for that system state and the values of its leaf states. This system of equations is more complicated to set up, but it is also much smaller and easier to solve.

Each equation in the second system of equations takes the following form,

$$V(s) = g(s, \delta(s)) + \sum_{s' \in S} W(s'|s, \delta(s))V(s'), \quad (4.1)$$

where $\delta(s)$ denotes the policy tree for state s and the variable s' denotes a system state at a leaf of this tree. This is a generalization of the equation used to evaluate a conventional, single-step policy. The function $g(s, \delta(s))$ generalizes the reward function from the single-step to the multistep case. It represents the expected reward for

executing policy tree $\delta(s)$, not including the terminal values backed-up from the leaf nodes of the policy tree.

The function W is used to weight the unknown values of the system states at the leaves of the policy tree. In general, these unknown values are weighted by both the discount factor and the probability of reaching state s' from the root of the policy tree. The weight vector W has one entry for each system state, with each entry defined as follows:

$$W(s'|s, \delta(s)) = \sum_{l_i \in \{l \in L | l = s'\}} Pr(l_i|s, \delta(s)) \beta^{\text{depth}(l_i)}. \quad (4.2)$$

In this definition, L denotes the set of all leaves of the policy tree, $\{l \in L | l = s'\}$ denotes the subset of leaves that are equal to system state s' , l_i denotes a particular leaf of the policy tree, $Pr(l_i|s, \delta(s))$ denotes the probability of reaching this leaf from the root, and $\text{depth}(l_i)$ denotes the depth of leaf l_i in the policy tree.

These weights can be computed in a simple depth-first traversal of the policy tree that calculates the probability of reaching each leaf, discounts this probability by the depth of the leaf in the policy tree, and adds the discounted probability to an accumulator for the corresponding system state. Thus, the complexity of computing the weight vector is linear in the number of nodes in the policy tree. But, in return, it reduces the complexity of policy evaluation from $O(m^3)$, where m is the number of information states visited by the policy, to $O(|S|^3)$, where $|S|$ can be dramatically less than m .

Policy improvement

The policy-improvement step consists of performing a multistep backup for each system state and, if an improved policy tree is found, changing the policy accordingly. I will assume that AO* is used to perform multistep backups. However, other search algorithms can be used and the structure of a problem, or the need to compute a

solution quickly, may make another search algorithm a better choice. For example, some POMDPs have an action set that consists of two types of actions: actions that provide perfect information and actions that provide no information. This makes the search tree for a multistep backup an OR tree, instead of an AND/OR tree, and a policy tree is simply an open-loop sequence of actions corresponding to a path through the OR tree. For such problems, A* or related branch-and-bound algorithms can be used to perform a multistep backup more efficiently than search algorithms for AND/OR trees. Section 4.3.2 describes an example for which this search strategy is more appropriate.

Because a heuristic-search algorithm that searches for the best policy tree can be computationally intensive, it may sometimes be advantageous to perform multistep backups using a more efficient search that may not find the best policy tree. For example, a greedy search can perform a multistep backup very quickly. It may not find the best policy tree or always recognize improvement when it is possible, but it may improve a multistep policy at a faster rate than a more complete search.

Like other best-first search algorithms, AO* relies on a search heuristic h to direct the search and prune the search space. If h provides an upper bound on the value of each information state π , it is said to be admissible. Given an admissible heuristic, AO* will find the best policy tree up to the depth bound using the current value function to evaluate terminal states.

A commonly used admissible heuristic for POMDPs is the optimal value function for a completely observable version of the same problem. Let V_{CO}^* denote this value function. An upper bound on the optimal value of any information state π is $\sum_{s \in S} \pi(s) V_{CO}^*(s)$. Proof that this is an upper bound is based on Jensen's inequality and can be found in several places [3].

Instead of computing V_{CO}^* and using it to estimate the value of an information state, what if we use the current multistep value function V in its place? Because

$V(s) < V_{CO}^*(s)$ for every system state s , this search heuristic will prune more aggressively and improve the efficiency of the search. However it is a lower — not an upper — bound on the optimal value function, and that means it can underestimate the optimal value of an information state. In other words, it is not an admissible heuristic.

A surprising and useful result is that using the current value function to estimate the value of information states guarantees convergence to an optimal multistep policy, even though it is not an admissible heuristic. Multistep policy improvement using this inadmissible heuristic may not always find the best policy tree for each system state, based on the current value function and depth bound. However, it always finds an improved policy when the current policy is not optimal. For an iterative algorithm such as policy iteration, this is sufficient to ensure eventual convergence to an optimal policy.

Theorem 4.1 *If the current value function is used as a search heuristic for multistep backups in the policy-improvement step, an improved policy is found whenever the current policy is not optimal.*

Proof. Because the policy tree for a state is not changed unless a policy tree with a greater backed-up value is found, it is immediate that the backed-up value for every state after policy improvement is greater than or equal to the current state value. To show that an improved policy is found whenever the current policy is suboptimal, we need to show that, for at least one state, a new policy tree is found with a backed-up value that is greater than the value of that state under the current policy. By Howard's policy-improvement theorem, it follows that the value of every state under the new policy (after policy evaluation) must be as great or greater than its value under the current policy — and greater for at least one state.

Proof that an improved policy tree is found for at least one state is by induction on the depth of policy trees. If the value of any state can be improved by a policy tree

of depth one, that is, by taking a single action that resets memory, then an improved policy is found because each policy tree of depth one is evaluated when the root node is expanded. Now make the inductive hypothesis that an improved policy is found if the value of any state can be improved by a policy tree of depth less than k . We show that an improved policy must be found if the value of any state can be improved by a policy tree of depth k .

Suppose there is a policy tree of depth k that improves the value of a state s . If it is evaluated in the course of the search, an improved policy is found and the theorem follows. If it is not, there must be some partial policy tree of this improved policy tree that has a value less than the current policy tree. Otherwise, the improved policy tree would have been evaluated. Therefore, the value of some unexpanded node of this partial policy tree is underestimated by the search heuristic. But, if that is the case, then the current value of some system state s' can be improved by a policy tree of depth less than k . By the inductive hypothesis, it follows that policy improvement finds an improved policy. \square

Convergence

Bounding the depth of a policy tree, that is, bounding the interval between one action that provides perfect information and the next, ensures that the policy-improvement step terminates each iteration. It also keeps the number of possible policies finite and, from this, it follows that multistep policy iteration converges after a finite number of iterations. If the search heuristic is admissible (or has the special property established by Theorem 4.1), it converges after a finite number of iterations to a policy that is optimal among all policies with this bound on the interval between actions that provide perfect information. This raises the question: can multistep policy iteration find a policy that is optimal without any bound on the interval between one action that provides perfect information and the next?

If policy iteration converges and the depth bound is used to prune any search path during the last iteration, the policy to which it converges may be improved by waiting longer before taking an action that acquires perfect information. But if policy iteration converges without invoking the depth bound to prune any search path during the last iteration, the policy to which it converges must be optimal without bound. This suggests a modification of the standard convergence test for policy iteration. If policy iteration converges by the standard criterion (that is, the policy is unchanged from one iteration to the next) but uses the depth bound to prune some search path during the last iteration, the algorithm is continued with an increased depth bound. Two conditions must now be satisfied for convergence.

1. the policy does not change from one iteration to the next, and
2. the depth bound is not used to prune any search path during the last iteration.

Given this modified test for convergence, it is possible to prove convergence to an optimal policy under an additional assumption.

Theorem 4.2 *Multistep policy iteration converges to an optimal policy after a finite number of iterations if every optimal policy acquires perfect information at finite intervals.*

Proof. Because there is no limit to how far the depth bound can be increased and because every system state has a finite policy tree that is optimal, there is some depth bound such that an optimal multistep policy and value function for policy trees up to that depth bound is also optimal for policy trees of unbounded depth. What we must show is that given a value function that is optimal for policy trees of unbounded depth, there is some finite depth bound such that all search paths are pruned without invoking the depth bound. The proof is by contradiction.

Assume that for some system state, there is no depth bound such that all search paths are pruned. It follows that there is an infinite number of partial policy trees

with a value greater than the value of an optimal policy tree. Therefore, there is some partial policy tree of infinite depth with a value that is equal to or greater than the value of an optimal policy tree. But, this contradicts the assumption that there is no optimal policy that does not necessarily acquire perfect information at finite intervals. \square

There may be more than one optimal policy. If there is an optimal policy that acquires perfect information at finite intervals and another optimal policy that does not, multistep policy iteration will not necessarily converge after a finite number of iterations.

4.3 Performance

To convey a better understanding of how multistep policy iteration works, this section describes how it performs on two concrete examples.

4.3.1 Machine maintenance

The following example was first described by Smallwood and Sondik [102] and is representative of a wide class of maintenance problems studied in the operations-research literature. Smallwood and Sondik used it to illustrate a finite-horizon POMDP. Platzman [90] used the same example to illustrate an infinite-horizon POMDP with an average reward performance criterion. Here, we consider the same example with a discounted infinite-horizon performance criterion.

Example 4.1 (*Smallwood and Sondik [102]*). Consider a machine that manufactures a product each time step. (A time step corresponds to a production cycle). The machine has two internal components that can fail. Failure of one or both internal components may cause the machine to manufacture defective products.

The machine can be modeled by a three-state discrete-time Markov process in which the three states correspond to zero, one, or both components having failed. Each

internal component of the machine fails stochastically, with an independent failure probability of 0.1 each time step the machine is in operation. Once a component has failed, it remains in the failed state until maintenance is performed and the part is replaced.

The internal state of the machine affects the manufacturing process as follows. If neither internal component has failed, manufactured products will not be defective. If one internal component has failed, each manufactured product will be defective with probability 0.5. If both internal components have failed, each manufactured product will be defective with probability 0.75. Because the reward for manufacturing a good product is one unit and no reward is received for manufacturing a defective product, these probabilities determine the one-step expected reward.

Each time step (production cycle), one of four possible actions can be taken.

- **manufacture (M):** *Manufacture a product but do not examine it to determine whether it is defective or not.*
- **examine (E):** *Manufacture a product and examine it at a cost of 0.25 units. Examination reveals whether the product is defective (d) or not, and this provides imperfect information about the internal state of the machine.*
- **inspect (I):** *Dismantle the machine for one time step (which means nothing can be manufactured) and inspect its internal components at a cost of 0.5 units. This provides perfect information about the internal state of the machine. Any component that has failed is replaced, and replacement incurs an additional cost of one unit per replaced component.*
- **replace (R):** *Dismantle the machine for one time step and replace both internal components at a cost of one unit each, without first performing a costly inspection to determine whether either component has actually failed.*

The problem is to find an optimal policy for operating, inspecting and maintaining the machine. The performance criterion is infinite-horizon discounted return and the discount factor is 0.99.

In this example, there are two actions that reset memory; the inspect and replace actions both restore the machine to a state of perfect working order. Therefore, the multistep approach can be applied to this problem. Using the same parameters as Smallwood and Sondik [102], an optimal multistep policy is to manufacture for eight time steps without examining the product and then to inspect the machine and replace any internal component that has failed. This assumes the the decision maker knows the initial state of the machine and it is in perfect working order. If the machine has one failed component, the optimal action is to inspect to see which component has failed before replacing it. If the machine has two failed components, the optimal action is to replace both without first inspecting them. Figure 4.1 shows the finite-state controller that corresponds to this policy.

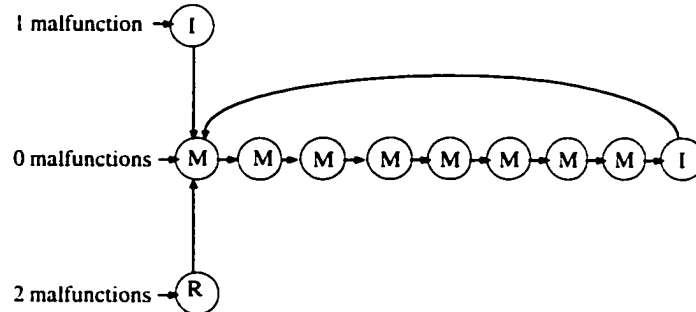


Figure 4.1. Optimal multistep policy for maintenance problem.

If the cost of the examine action is reduced from 0.25 to 0.1 units, a more interesting policy that includes branching is found. The policy shown in Figure 4.2 is optimal when the depth bound for multistep lookahead is 16. It is not optimal in general, however, and in Section 7.2.2, I show that multistep policy iteration cannot converge to an optimal policy for this problem with these parameter settings.

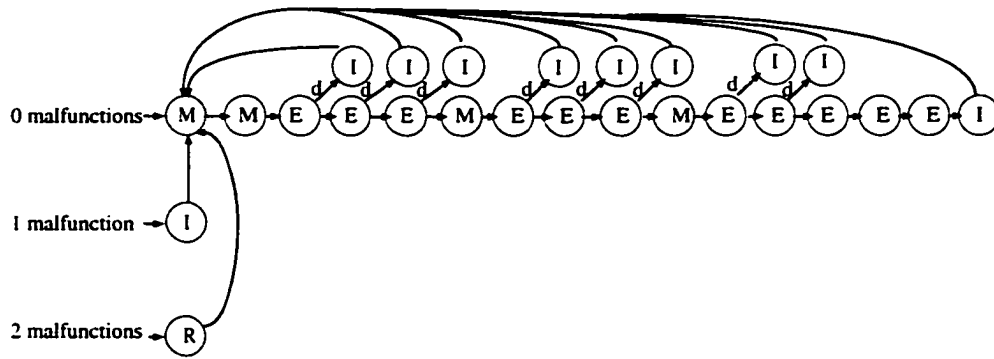


Figure 4.2. Multistep policy for maintenance problem with changed parameters.

This example suggests that it is more likely for multistep policy iteration to converge to an optimal policy when an optimal policy contains only two types of actions; actions that provide perfect information and actions that provide no information. Actions that provide imperfect information introduce branching and it may be difficult to prune every branch of the policy tree without invoking the depth bound. Multistep policy iteration can still find arbitrarily good policies in this case, of course. But, it seems particularly well-suited for problems without branching, such as the following example.

4.3.2 Gridworld

Artificial intelligence researchers often use simple “gridworld” problems to study algorithms for planning and learning problems that are modeled as Markov decision processes. Although close in spirit to path-planning problems for mobile robot navigation, these are toy problems that are only intended to provide a test-bed for examining the behavior of algorithms. The following example helps to give a better sense of the size of problems that can be solved using multistep policy iteration. It also provides an opportunity to test its performance using different search heuristics.

Example 4.2 *A robot in the gridworld shown in Figure 4.3 must find its way to a goal location, which is in the upper left-hand corner of the grid. Each cell of the grid corresponds to a state, with the states numbered for convenient reference. The robot*

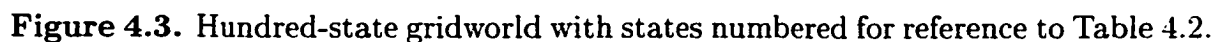


Table 4.2. Optimal multistep policy for gridworld of Figure 4.3.

has four actions it can take to move about the grid: it can move north (N), south (S), east (S), or west (W), one cell at a time. The robot can also halt (H) and it receives a large reward for halting in the goal state.

To complicate this path-planning problem, the robot's actions have unpredictable effects. If the robot attempts to move in a particular direction, it succeeds with probability 0.8. With probability 0.05 it moves in a direction that is 90 degrees to one side of its intended direction, with probability 0.05 it moves in a direction that is 90 degrees to the other side of its intended direction, and with probability 0.1 it does not move at all. If the robot's movement would take it outside the grid, it remains in the same state. The actions the robot takes to move about the grid provide no information, but it can choose an observe action (O) that perfectly reveals its current state. By interleaving the observe action with move actions, the robot can monitor its progress toward the goal. Each move action incurs a cost of 4, creating an incentive to reach the goal by as direct a route as possible. The observe action incurs a cost of 1. If the robot halts in the goal state, it receives a reward of 1000; if it halts in any other state, it incurs a cost of 10.

The gridworld shown in Figure 4.3 has 100 states and Table 4.2 shows the optimal multistep policy. It specifies, for each state, an open-loop sequence of actions that ends with observation of the current state. Note that the interval between observations varies from state to state in an intuitively reasonable way. Multistep policy iteration converges to this policy in 24 seconds, not much longer than it takes to converge to an optimal policy for the completely observable equivalent of this problem for which there is no cost for making an observation.

To test how the algorithm's efficiency is affected by the size of the state set, we ran it on a series of similar gridworlds of increasing size. Table 4.3 compares how long it takes conventional policy iteration to converge, assuming each move action provides perfect information, to how long it takes multistep policy iteration to converge to an

optimal policy for interleaving move actions with costly observations. It suggests the efficiency of this algorithm, relative to the efficiency of conventional policy iteration, is insensitive to the size of the state set. The additional time it takes multistep policy iteration to converge remains a small constant factor of the time it takes conventional policy iteration to converge.

Number of states in gridworld	100	200	300	400	500
Standard policy iteration	7	66	199	508	1262
Multi-step policy iteration	24	286	711	1540	2964

Table 4.3. Time to convergence in seconds as problem size increases.

This is not a surprising result. The complexity of evaluating a multistep policy depends on the number of system states only. The complexity of improving a multistep policy is equal to the number of system states times the complexity of performing a multistep backup, and the complexity of performing a multistep backup does not grow as the number of system states grows. Of course, the time it takes both conventional and multistep policy iteration to converge begins to increase dramatically as the size of the state set grows because of the increased expense of policy evaluation. Multistep value iteration is likely to outperform multistep policy iteration for problems with large state sets.

Although insensitive to the size of the state set, the efficiency with which a multistep backup can be performed is very sensitive to other factors. This can be demonstrated by varying the cost of the observe action in the 100-state gridworld example. If the cost is zero, it is optimal to observe after every move action and the problem is equivalent to a completely observable MDP and easily solved. As the cost of the observe action increases, the time it takes to perform a multistep backup, and consequently the time it takes for multistep policy iteration to converge, begins to increase dramatically, as Table 4.4 shows.

Cost for observing state	0.0	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5
Time to converge in seconds	7	17	24	28	40	90	276	533	1087	> 10000

Table 4.4. Algorithm efficiency as cost for observation increases.

In part, these results reflect the fact that as the cost of the observe action increases, the robot tends to take longer open-loop sequences of actions before observing. However, this effect is rather minor. The length of the action sequences taken by the robot increases by a relatively small amount as the cost of the observe action increases. The principal reason for the increase in the time it takes to converge is that increasing the cost of the observe action makes the search heuristic less reliable. The search heuristic estimates the value of an information state by assuming that perfect information can be acquired at no cost. As the cost of the observe action increases, the assumption that perfect information is free becomes less reliable and the heuristic prunes less and less of the search space. This reflects a well-known principle of heuristic search: the less reliable a heuristic, the less efficient the search.

This effect can also be observed by comparing the use of the current multistep value function as a search heuristic, which we used to obtain these results, with use of the optimal value function for the completely observable version of this problem as a search heuristic, which is admissible but prunes less aggressively. For the gridworld of Figure 4.3 with an observation cost of 1.0, multistep policy iteration using the current value function as a search heuristic converges in 24 seconds. Using the optimal value function for the completely observed version of the problem as a search heuristic, a commonly-used search heuristic for POMDPs, it takes more than sixteen hours for the same algorithm to converge! This vividly demonstrates two things. First, it shows just how much the efficiency of heuristic search depends on having a good heuristic. Second, it demonstrates just how effective the heuristic introduced in Section 4.2 is and it underlines how remarkable it is that a non-admissible heuristic that prunes so aggressively can also ensure convergence to an optimal policy.

Nevertheless, Table 4.4 shows that there are some cases in which even this search heuristic is ineffective. There are two ways to speed up search in this case. One possibility is to accept a suboptimal result in exchange for improved efficiency. One approach is to use greedy search in multistep backups [33]. In practice, policies found using greedy multistep backups are close to optimal, at least for gridworlds such as this.

A second possibility is to find a more powerful search heuristic that still guarantees convergence to an optimal policy. Our search heuristic estimates the value of information states using a value function that is defined only for the system states. We might call this a “value of perfect information heuristic” because it estimates the value of an information state by assuming that perfect information can be obtained at no cost. A more reliable search heuristic might use a value function that is defined for other points in information space as well. There has been work on computing upper bounds on the optimal value function for a fixed or variable grid of points in information space [38, 10]. By the convexity property of the optimal value function for a POMDP (or equivalently, by the non-negativity of the value of information), an upper bound on the value of any point in information space can be found by interpolation using such a grid of information states. We might call this a “value of imperfect information heuristic” because it estimates the value of an information state by assuming that imperfect information can be obtained at no cost — an assumption that introduces less inaccuracy into the search heuristic because it assumes less free information.

Defining a value function for a finite grid of points in information space makes this heuristic more expensive to compute and there is likely to be a tradeoff between

time spent computing a more accurate heuristic and the resulting reduction in search complexity. Experimentation can find the best balance.²

To summarize, these gridworld experiments show that multistep policy iteration can be very efficient. Because it defines a value function for the system states only, it can solve problems that approach the size of problems that can be solved by conventional policy iteration. Because it relies on heuristic search to perform backups, however, the efficiency with which it can find optimal policies depends crucially on the power of the search heuristic it uses.

4.4 Discussion

This chapter shows that any POMDP with an action set that includes one or more actions that reset memory can be solved using an approach that combines a dynamic-programming algorithm for completely observable MDPs (such as policy iteration or value iteration) with a heuristic-search algorithm for decision trees (such as AO*). I have focused on the case in which the dynamic-programming algorithm is policy iteration and the actions that reset memory provide perfect information.

There is an interesting relationship between the multistep approach described here and the concept of a semi-Markov MDP. In a semi-Markov MDP, state transitions occur at stochastic intervals of time and the actions that cause state transitions can sometimes be viewed as *macros*, that is, open-loop or closed-loop policies that control the process for some period of time before relinquishing control to a higher-level policy. From this perspective, a multistep policy can be viewed as the solution of a semi-Markov MDP and the policy trees it specifies for each system state can be viewed as macros that solve finite POMDPs for which a start state and terminal states are given. This perspective relates multistep dynamic programming to recent

²Information states visited by the current multistep policy may provide a useful grid of points, particularly since their values have already been computed.

work on using semi-Markov MDPs for planning at multiple time scales [107, 41]. The multistep approach differs by using heuristic search to find macros and by adapting this framework to solve POMDPs.

The model described in this chapter was originally developed for a subset of POMDPs for which observations provide perfect information but incur a cost, a subset of POMDPs illustrated by the gridworld example. For such problems, a version of multistep policy iteration that uses greedy search to perform multistep backups is described in [33]. A related version of the algorithm that ensures convergence to an optimal multistep policy is described in [34]. This chapter presents this research in its most general form. A reinforcement-learning algorithm for solving a related subset of problems is described in [36].

The intermediate POMDP model created by allowing observations to have a cost, but which still assumes observations provide perfect information, can be solved much more easily than the general POMDP problem. Multistep dynamic programming quickly finds optimal policies for problems with hundreds of system states, and good policies may be found for problems with thousands of system states. POMDPs that do not include an action that periodically resets memory are far more difficult to solve, even when they have a mere handful of states. Thus, one of the contributions of this chapter is to identify a special case of the POMDP problem that is more tractable than the problem in its general form.

Before considering the POMDP problem in its general form, we consider another way to combine dynamic programming for infinite-horizon MDPs and AO* that makes it possible to solve a larger subset of POMDPs than can be solved using the multistep approach.

CHAPTER 5

HEURISTIC SEARCH IN CYCLIC DECISION GRAPHS

Acyclic sequential decision problems that can be solved by dynamic programming can usually be solved more efficiently by heuristic search, given a start state and a heuristic evaluation function. However, heuristic-search algorithms have not been developed for cyclic sequential decision problems, such as infinite-horizon MDPs. The multistep approach of the previous chapter finessed this by using heuristic search to solve a set of finite search problems, one for each system state, and combined these solutions into an overall solution of an infinite-horizon MDP, using dynamic programming. This chapter introduces a generalization of heuristic search that can solve infinite-horizon MDPs directly, including problems the multistep approach cannot solve. It shares the advantage of all heuristic-search algorithms over dynamic programming: for a given start state, it can find an optimal policy without evaluating the entire state space.

5.1 An example the multistep approach cannot solve

The multistep approach of the previous chapter finds a finite-state controller of a specific type I called a multistep policy. Each cycle in a multistep policy includes an action that resets memory (for example, by providing perfect information) and this imposes a finite bound on the interval between one action that resets memory and the next.

However, it is possible for a policy to visit a finite number of information states without necessarily resetting memory at finite intervals. Such a policy corresponds

to a finite-state controller with a cycle that does not include an action that resets memory. This possibility is best illustrated by an example.

Example 5.1 (*Cassandra, Kaelbling and Littman [14]*). *An agent is standing in front of two closed doors. Behind one door is a tiger and behind the other is a reward. The two possibilities — the tiger is behind the door on the left or the tiger is behind the door on the right — represent two possible states.*

Opening the door with the tiger behind it results in a penalty of 100. Opening the door with the reward behind it results in a reward of 10. Before opening one of the two doors, the agent can listen for the tiger. However, listening is neither free nor reliable. Each time the agent listens, it incurs a cost of 1. If it hears the tiger behind the door on the left (TL), there is a probability of 0.15 that the tiger is actually behind the door on the right. Hearing the tiger behind the door on the right (TR) has the same probability of error. The agent can listen any number of times before opening one of the doors.

The problem is to decide how long to continue listening before opening a door, and which door to open. After the agent opens a door and receives the reward or penalty behind it, the problem is reset by closing the door and randomly placing the tiger behind one of the two doors. Repeating the problem each time the door is opened makes this an infinite-horizon problem for which we assume a discount factor of 0.95.

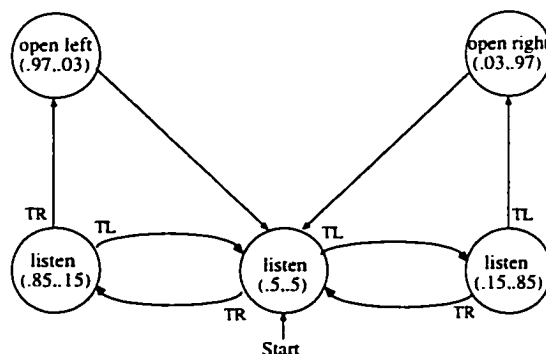


Figure 5.1. Optimal finite-state controller for Example 5.1.

Figure 5.1 shows an optimal finite-state controller for this example, assuming the starting information state is a uniform probability distribution over the two possible states of the system. (Later in this chapter we will see how this optimal policy can be computed.) Each memory state is labeled by an action and by the unique information state that corresponds to it; the information state gives the probability that the tiger is behind the door on the left followed by the probability that it is behind on the door on the right. Including the starting information state, the policy visits only five information states over an infinite horizon. Opening either door resets memory to the starting information state.

In many respects, this example is similar to those solved by the multistep approach of the previous chapter; it has an action that resets memory (opening a door), each memory state corresponds to a unique information state, and an optimal policy visits a finite number of information states. There is a significant difference, however. There is no bound on the interval at which memory is reset by opening a door. The agent can listen indefinitely, depending on what it hears, before opening a door. The listen action creates a loop back to a previously visited information state, without resetting memory. It doesn't reset memory, in the sense we defined, because the posterior information state of the listen action depends on the prior information state. Thus, an optimal finite-memory policy for this problem has a more complex structure than multistep policy iteration can find using AO* to perform backups.

This chapter describes an algorithm that can solve the tiger problem and similar problems, in addition to any problem the multistep approach can solve. I call it LAO*, where "L" stands for "loops," because it is a generalization of AO* that can find solutions with loops. LAO* can find an optimal policy for an infinite-horizon POMDP for which an optimal policy visits a finite number of information states. Clearly, this is the best we can do given the constraint that each memory state of a finite-state controller corresponds to a unique information state.

5.2 LAO*

LAO* is a generalization of AO* that can find solutions with loops. It is such a natural generalization of AO* that it is difficult not to speculate why it has not been proposed before. I can think of three possible reasons.

The first is that solutions with loops do not make sense for problems for which AO* was originally developed, such as problem-reduction search and theorem proving. A loop in the solution to a problem-reduction problem means the problem cannot be successfully reduced to primitive subproblems. A loop in the solution to a theorem-proving problem represents circular reasoning. However, solutions with loops do make sense for problems that can be formalized as infinite-horizon MDPs.

A second reason is that solutions with loops can have infinite value (or infinite cost) and thus appear to be ill-defined. However, this issue is addressed in the literature on infinite-horizon MDPs and it is straightforward to adopt the same solution. For example, a simple discount factor ensures that the value (or cost) of a policy with loops is finite.

A third reason is that the presence of loops destroys the partial ordering of state values on which the dynamic-programming step of AO* relies. But the dynamic-programming step is easily generalized to allow solutions with loops by replacing the backwards-induction algorithm used by AO* with a dynamic-programming algorithm for infinite-horizon MDPs such as policy iteration or value iteration. This simple generalization creates a heuristic-search algorithm that can find solutions with loops.

Table 5.1 summarizes LAO* and shows how closely it is related to AO*. The forward-search step of LAO* identifies the best partial policy graph and expands it, in the same manner as the forward-search step of AO*. LAO* generalizes the graph-search version of AO* because both must recognize when the successor of an expanded node is already part of the explicit graph. The graph-search version of AO*

1. *Input*: A starting information state.
2. *Initialization*: The explicit graph initially consists of a root node that corresponds to the starting information state.
3. *Forward search*. Expand the best partial policy graph as follows:
 - (a) *Identify best partial solution*. From the start node, traverse the best partial policy graph to identify nonterminal tip states that are eligible to be expanded.
 - (b) *Termination test*. If there are no unexpanded states in the best partial policy graph, go to step 5.
 - (c) *Expand partial solution*. Expand some nonterminal tip state of the best partial policy graph and add its outgoing arcs and successor states to the explicit graph.
4. *Dynamic programming*. To update state values, create a set Z that contains the expanded state and all of its ancestors in the explicit graph. Perform policy iteration on the states in set Z until convergence. Go to step 3.
5. *Output*: An optimal policy graph.

Table 5.1. LAO*.

allows the same state to be reached along different branches of the search graph but does not allow cycles. LAO* allows both.

Like AO*, LAO* performs dynamic programming on the expanded state and its ancestors in the explicit graph. It differs from AO* by using a dynamic-programming algorithm for infinite-horizon MDPs to cope with loops. For now, I assume that policy iteration is used because it converges to exact state values. (At the end of this section, I discuss the possibility of using value iteration.) Because policy iteration is initialized with current state values, it may converge quickly. Nevertheless, it is a more complex and time-consuming algorithm than the backwards-induction algorithm used by AO*. This increase in complexity reflects the difficulty of handling solutions with loops.

Performing policy iteration on the expanded state and its ancestors may change the best action for some states, and thereby change the best partial policy graph. The same is true for AO* and, as a result, not every state in the current best partial policy

graph may be reachable from the start state at the end of the dynamic-programming step. Nevertheless, policy iteration must be performed on all ancestor states until convergence to ensure that all states in the explicit graph have exact, admissible values. But, like AO*, dynamic programming may not need to be performed on the entire explicit graph.

LAO* has the same properties as AO* and other heuristic-search algorithms. Given an admissible heuristic evaluation function, backed-up state values at the end of the dynamic-programming step are admissible and LAO* converges to an optimal policy without (necessarily) evaluating the entire state space.

Theorem 5.1 *If the heuristic evaluation function h is admissible and policy iteration is used to perform the dynamic-programming step of LAO*, then:*

1. $f(\pi) \geq f^*(\pi)$ for every state π of the explicit graph, after each iteration
2. $f(\pi_0) = f^*(\pi_0)$ for the start state π_0 , when LAO* terminates.

Proof: (1) For every terminal state of the explicit graph, $f(\pi) = f^*(\pi)$ by definition. For every nonterminal tip state π of the explicit graph, $f(\pi) = h(\pi) \geq f^*(\pi)$ by the admissibility of the heuristic evaluation function. Proof that the values of all non-tip states are admissible after each iteration of LAO* is by induction. Clearly, it is true at the start of the algorithm. We make the inductive assumption that it is true at the beginning of some iteration. It remains true for states that are not ancestors of the expanded state because their values are not changed. Policy iteration is performed until convergence on the expanded state and its ancestors in the explicit graph. If the values of all tip states of the explicit graph are optimal, then the other states must converge to their optimal values by the convergence proof for policy iteration. Because the values of all tip states are admissible, all states must converge to values that are as good or better than optimal, in other words, to admissible values.

(2) The search algorithm terminates when the best policy graph rooted at π_0 has no unexpanded states. It is contradictory to suppose $f(\pi_0) > f^*(\pi_0)$ since that implies a complete solution that is better than optimal. By (1) we know that $f(\pi) \geq f^*(\pi)$ for every state in the explicit graph. Therefore, $f(\pi_0) = f^*(\pi_0)$. \square

It is obvious that LAO* terminates after a finite number of iterations if the implicit graph is finite. If the implicit graph is infinite, as it is for infinite-horizon POMDPs, it terminates after a finite number of iterations if the number of finite-size partial policy graphs for which $f(\pi_0) \geq f^*(\pi_0)$ is finite. This holds under the hypothesis of the following theorem.

Theorem 5.2 *If no optimal policy can visit an infinite number of information states, beginning from π_0 , then LAO* terminates after a finite number of iterations.*

Proof: The proof is by contradiction. The algorithm fails to terminate after a finite number of iterations if and only if the number of finite-size partial policy graphs for which $f(\pi_0) > f^*(\pi_0)$ is infinite. If the number is infinite, then it is impossible to bound the size of a partial policy graph for which $f(\pi_0) > f^*(\pi_0)$ and there must be an infinite path through the implicit graph, starting from π_0 , such that $f(\pi) > f^*(\pi)$ for every information state π on this path. But, this is only possible if there is an optimal policy that visits an infinite number of information states. \square

There may be more than one optimal policy graph. If there is an optimal policy graph that visits a finite number of information states and another optimal policy graph that visits an infinite number of information states, LAO* will not necessarily converge after a finite number of steps.

Use of policy iteration in the dynamic-programming step of LAO* becomes computationally prohibitive as the number of state values that need to be updated grows. The complexity of the backwards-induction algorithm used by AO* is linear in the number of state values that need to be updated. The complexity of the policy-

evaluation step of policy iteration is cubic in this number of states, and more than one iteration may be needed for it to converge.

An alternative is to use value iteration to perform the dynamic-programming step of LAO*. A single iteration means performing a single backup for each state, which is computationally equivalent to the backwards-induction algorithm used by AO*. The difference is that the presence of loops means backed-up values are no longer exact. This means the forward-search step of LAO* is no longer guaranteed to identify the best partial policy graph and to expand nodes in a best-first order. This disadvantage may be more than offset by the improved efficiency of the dynamic-programming step of the algorithm. However, it complicates the algorithm. I will not describe here how LAO* changes when value iteration is used to perform the dynamic-programming step or how these changes affect the convergence proofs.

5.3 Related work: Completely observable MDPs

I have presented LAO* as an approach to solving a subset of infinite-horizon POMDPs. However, there are other problems for which a policy may contain loops, in particular, infinite-horizon completely observable MDPs. Because LAO* can find an optimal policy without evaluating the entire state space, it may solve such problems more efficiently than value iteration or policy iteration. In fact, algorithms very similar to LAO* have recently been developed in the AI community for planning problems formalized as completely observable infinite-horizon MDPs.

Barto, Bradtke, and Singh [4] describe an algorithm called real-time dynamic programming (RTDP) that generalizes Korf's [56] learning real-time heuristic search algorithm (LRTA*) from deterministic shortest-path problems to stochastic shortest-path problems, or, in general, to MDPs. They show that under certain conditions, RTDP converges to an optimal solution without evaluating the entire state space. This parallels the principal result of this chapter; in fact, LAO* and RTDP solve the

same class of problems. The difference is that RTDP relies on trial-based exploration — a concept adopted from reinforcement learning — to explore the state space and determine the order in which to update state values. By contrast, LAO* finds a solution by systematically expanding a search graph in the manner of heuristic-search algorithms such as A* and AO*.

Dean *et al.* [21] describe a related algorithm that performs policy iteration on a subset of the states of an MDP, using various methods to identify the most relevant states and gradually increasing the subset until eventual convergence (or until the algorithm is stopped). The subset of states is called an *envelope* and a policy defined on this subset of states is called a *partial policy*. Adding states to an envelope is very similar to expanding a partial solution in a search graph and the idea of using a heuristic to evaluate the fringe states of an envelope has also been explored [109, 23]. However, this algorithm is presented as a modification of policy iteration (and value iteration), rather than a generalization of heuristic search. In particular, the assumption is explicitly made that convergence to an optimal policy requires evaluating the entire state space.

Both of these algorithms are motivated by the problem of search (or planning) in real-time and both allow it to be interleaved with execution. The time constraint on search is often the time before the next action needs to be performed and decision-theoretic approaches to optimizing the value of search in the interval between actions have been explored [21, 109]. Therefore, these algorithms can be viewed as real-time counterparts of LAO*, much as RTA* and LRTA* [56] are real-time counterparts of A* for deterministic search problems, and LAO* fills a gap in the taxonomy of search algorithms.

RTDP and the related envelope approach to policy and value iteration represent a solution as a mapping from states to actions, albeit an incomplete mapping called a partial policy; this reflects their derivation from dynamic programming. LAO*

represents a solution as a cyclic graph (or equivalently, a finite-state controller), a representation that generalizes the graphical representations of a solution used by search algorithms like A^* (a simple path) and AO^* (an acyclic graph); this reflects its derivation from heuristic search. The advantage of representing a solution in the form of a graph is that it exhibits reachability among states explicitly and makes analysis of reachability easier. A description of LAO^* for completely observable MDPs is given in [37].

5.4 Discussion

LAO^* has been implemented and quickly (within less than a second) converges to an optimal policy for the tiger problem described at the beginning of this chapter. It is unclear whether the set of problems LAO^* can solve and the multistep approach cannot is large or significant. However, the fact that it converges to an optimal policy whenever an optimal policy visits a finite number of information states is an intellectually pleasing result. Clearly, it is the most we can hope for given the constraint that each memory state corresponds to a unique information state.

Both LAO^* and multistep dynamic programming combine dynamic programming for infinite-horizon MDPs with the heuristic-search algorithm AO^* , but they do so in different and complementary ways. Multistep dynamic programming identifies a simple search step inside policy iteration and value iteration — the single-step backup — and replaces it with a more general search algorithm that does multistep lookahead. LAO^* identifies a simple dynamic-programming algorithm inside AO^* — backwards induction — and replaces it with a more general dynamic-programming algorithm that can cope with loops. LAO^* is more general than multistep dynamic programming; it can find an optimal policy without assuming that memory is reset at finite intervals. However, both algorithms are limited to finding policies that visit

a finite number of states. The chapters that follow present a more general approach to finite-memory control of POMDPs that does not share this limitation.

CHAPTER 6

DYNAMIC PROGRAMMING FOR POMDPS

This is a transitional chapter that does not present new results. Instead, it reviews previous work that provides the foundation for the algorithms described in the rest of this thesis. These algorithms differ from the algorithms described in chapters 4 and 5 by representing the value function in a different, and more general, way. This chapter reviews this more general representation of the value function and some algorithms for solving POMDPs that use it.

The transformation of a POMDP into a completely observable MDP over information space makes it possible to define an optimality equation for POMDPs (see Equation 2.12) and to use dynamic programming to find an optimal value function and policy, at least in principle. The problem is that information space is continuous and there is an uncountably infinite number of possible information states. In the earlier chapters, this problem was finessed by considering policies that visit a finite number of information states. For such policies, the value function only needs to be defined, and updated, for a finite number of information states. This makes it possible to combine dynamic-programming algorithms for finite MDPs with heuristic search in order to solve special cases of the POMDP problem. However, this is not a general solution and most POMDPs cannot be solved by this approach.

This chapter describes how to represent a value function for all information states and how to perform dynamic programming using this more general representation of the value function. The results reviewed in this chapter, which have been developed over close to thirty years of research, make possible a general dynamic-programming

solution to the POMDP problem. The key step of dynamic-programming algorithms such as value iteration and policy iteration is computation of a new value function V^n based on a given value function V^{n-1} . For a POMDP that has been converted to a completely observable MDP over information space, this dynamic-programming update is defined as follows:

$$V^n(\pi) := \max_{a \in A} \left[r(\pi, a) + \beta \sum_{o \in O} Pr(o|\pi, a) V^{n-1}(T(\pi|a, o)) \right], \forall \pi \in \Pi. \quad (6.1)$$

It is not obvious that this definition can be translated into a practical algorithm because it requires evaluating the right-hand side expression for every one of an uncountably infinite number of information states.

The fundamental result that makes it possible to compute this dynamic-programming update is the proof by Smallwood and Sondik [102, 103] that it preserves the piecewise linearity and convexity of a value function. In other words, if V^{n-1} is a piecewise linear and convex function, so is V^n . Representation of the value function as a piecewise linear and convex function is the foundation for the dynamic-programming algorithms described in this chapter and the next.

6.1 Value-function representation

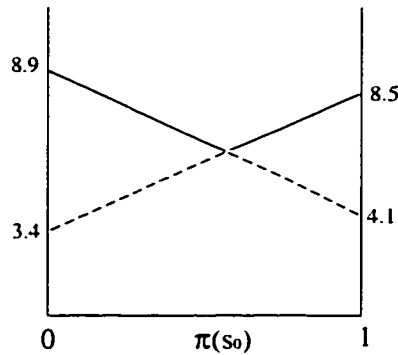


Figure 6.1. Example of a piecewise linear and convex value function for a two-state POMDP.

If a value function V defined for all of information space is piecewise linear and convex, it can be represented by a finite set Γ of real-valued $|S|$ -dimensional vectors. Figure 6.1 shows an example of a value function represented by two vectors, for a POMDP with two system states. Recall (from the discussion in Section 2.3) that for a two-state POMDP, an information state can be represented by a single number that represents the probability of being in one of the two states. (The probability of being in the other state is one minus this number.) The horizontal line at the bottom of Figure 6.1 represents the continuum of information states for a two-state POMDP by a line from 0 to 1 that represents the probability of being in state s_0 .

Each of the vectors in the value function shown in Figure 6.1 consists of two real numbers, one for each system state. The number for the first state is shown on the left and the number for the second state is shown on the right. A line connects numbers belonging to the same vector. The value of each information state π is defined as follows:

$$V(\pi) = \max_{\gamma \in \Gamma} \sum_{s \in S} \pi(s) \gamma(s). \quad (6.2)$$

In Figure 6.1, this formula corresponds to identifying the information state on the horizontal axis and drawing a straight line upwards to find the vector that optimizes its value. Thus, the upper surface of the vectors in Figure 6.1 represents the value function.

Astrom [3] first proved that an optimal value function is convex. This property has an intuitive explanation: information states that represent uncertainty about the system state are less valuable than information states of relative certainty. In other words, the convexity of the optimal value function is equivalent to the non-negativity of the value of information.

For finite-horizon POMDPs, Smallwood and Sondik [102, 103] proved that the optimal value function is piecewise linear, as well as convex. This is a corollary of their proof that the dynamic-programming update preserves the piecewise linearity

and convexity of the value function; it follows from this and the fact that the zero-horizon value function V^0 , a simple vector of zeros, is piecewise linear and convex.

Although the optimal value function for an infinite-horizon POMDP is always convex, it may or may not be piecewise linear. If it is not piecewise linear, Sondik [103, 104] shows (and we will too) that it can be approximated arbitrarily closely by a piecewise linear and convex function. Therefore, dynamic-programming algorithms for solving infinite-horizon POMDPs can also use a piecewise linear and convex representation of the value function.

6.2 Pruning

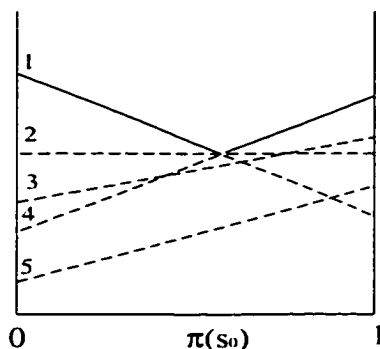


Figure 6.2. Non-minimal representation of a piecewise linear and convex value function for a two-state POMDP.

For every piecewise linear and convex value function, there is a minimal set of vectors that represents it [63]. Let Γ denote this minimal set. The same value function can also be represented by a non-minimal set of vectors. Figure 6.2 shows an example. Removing vectors 2, 3 and 5 will not change the value of any information state. Let $\tilde{\Gamma}$ denote a set of vectors that is a non-minimal representation of a value function. Vectors such as 2, 3 and 5 are said to be *dominated* vectors because there is no information state for which their value is not dominated by the value of some other vector.

Variables: $d, \pi(s)$ for all $s \in S$
Maximize: d
Constraints: $\pi \cdot \gamma - \pi \cdot \gamma^k - d \geq 0$ for all $\gamma^k \in \Gamma$
$\sum_{s \in S} \pi(s) = 1$
$\pi(s) \geq 0$ for all $s \in S$

Table 6.1. Linear program for testing vector dominance.

A set of vectors that includes dominated vectors can be pruned to yield a minimal representation of the value function. This pruning step plays a central role in dynamic-programming algorithms for POMDPs. The simplest method of pruning, first used by Eagle [27], is to prune any vector that is *pointwise dominated* by another. Pointwise dominance is a special case of dominance. A vector γ is pointwise dominated by another vector, γ' , if for each state s , $\gamma(s) \leq \gamma'(s)$. For example, vector 5 in figure 6.2 is pointwise dominated by vector 4. This method of pruning vectors is very fast but does not prune all dominated vectors. In Figure 6.2, both vectors 2 and 3 are dominated but neither is pointwise dominated.

Linear programming can always determine when a vector is dominated. For a given vector γ , the linear program shown in Table 6.1 tests whether adding it to a set of vectors Γ (which does not include γ) improves the value function represented by Γ for any information state. If it does, the variable d optimized by the linear program is the maximum amount by which the value function is improved and π is the information state that optimizes d . If it does not, that is, if $d \leq 0$, then γ is dominated.

This points to a straightforward algorithm for pruning a set of vectors $\tilde{\Gamma}$ to yield a minimal set Γ . The algorithm loops through $\tilde{\Gamma}$, removes each vector $\gamma \in \tilde{\Gamma}$, and solves the linear program with inputs γ and $\tilde{\Gamma} \setminus \{\gamma\}$ to test whether it is dominated. If it is not, γ must be part of the minimal set Γ and it is restored to $\tilde{\Gamma}$.

1. *Input:* A non-minimal set of vectors $\tilde{\Gamma}$ representing a value function.
2. Γ is initially empty.
3. Prune all pointwise dominated vectors from $\tilde{\Gamma}$.
4. Repeat until $\tilde{\Gamma}$ is empty:
 - (a) For some vector $\gamma \in \tilde{\Gamma}$, solve the linear program of Figure 6.1 with inputs γ and Γ .
 - (b) If γ is dominated, remove it from $\tilde{\Gamma}$.
 - (c) Otherwise find the vector $\gamma' \in \tilde{\Gamma}$ that optimizes the value of the information state π that solves the linear program. Remove γ' from $\tilde{\Gamma}$ and add it to Γ . (Note that γ remains in $\tilde{\Gamma}$.)
5. *Output:* A minimal set of vectors Γ representing the same value function.

Table 6.2. Algorithm for pruning a set of vectors.

A more efficient pruning technique is due to Lark [59, 120]. Instead of gradually pruning $\tilde{\Gamma}$ until it is a minimal set, it maintains two sets of vectors, $\tilde{\Gamma}$ and Γ , and gradually builds Γ by adding to it vectors from $\tilde{\Gamma}$ that are not dominated. A pruning algorithm that uses Lark's technique is outlined in Table 6.2. Both algorithms may solve the same number of linear programs. But Lark's approach is faster because the number of constraints in each linear program is bounded by $|\Gamma|$ instead of $|\tilde{\Gamma}|$, and $|\Gamma|$ is usually much smaller than $|\tilde{\Gamma}|$.

There is a subtlety in the implementation of Lark's algorithm that has been elucidated by Littman [63]. It is possible for more than one vector to optimize the value of a particular information state. For example, vectors 1, 2 and 4 in Figure 6.2 all optimize the information state at which they intersect. When Lark's algorithm searches in $\tilde{\Gamma}$ for a vector γ' that optimizes π and finds more than one, a tie-breaking rule must be invoked to choose one of these vectors to include in the minimal set. If the choice is made randomly, the set of vectors returned by Lark's algorithm may not be minimal. Littman shows that breaking ties based on lexicographic ordering

avoids this problem. (A vector γ is said to be *lexicographically greater* than another vector γ' if, given some predetermined ordering of the states in S , the first vector has a larger first component, or the two vectors are tied on their first i components and the first vectors is larger in component $i + 1$.)

6.3 Dynamic-programming update

The problem of computing the dynamic-programming update defined by Equation 6.1 can be expressed as follows: given a set of vectors Γ^{n-1} that represents a piecewise linear and convex value function V^{n-1} , find the minimal set of vectors Γ^n that represents the updated value function V^n .

Computing the dynamic-programming update is equivalent to performing a backup for all information states. It is straightforward to compute a backup for a single information state:

$$V^n(\pi) := \max_{a \in A} \left[r(\pi, a) + \beta \sum_{o \in O} Pr(o|\pi, a) V^{n-1}(T(\pi|a, o)) \right]. \quad (6.3)$$

When V^{n-1} is represented by a finite set of vectors, evaluating the expression on the right-hand side of this formula involves determining an action a , and for each observation o , a vector $\gamma \in \Gamma^{n-1}$, that optimizes the value of this expression. I will call this maximizing action and set of observation-vector pairs a *one-step policy choice* because it represents the best strategy for this information state based on one-step lookahead and the current value function V^{n-1} .

There is a close relationship between one-step policy choices and potential vectors in Γ . For any one-step policy choice, a vector $\gamma \in \Gamma^n$ can be constructed as follows:

$$\gamma_i^n(s) = r(s, \alpha(i)) + \beta \sum_{s' \in S, o \in O} Pr(s'|s, \alpha(i)) Pr(o|s', \alpha(i)) \gamma_{\tau(i,o)}^{n-1}(s'), \quad (6.4)$$

where i is the index of the vector being constructed, $\alpha(i)$ is the associated action, and $\tau(i, o)$ is the index of the successor vector in Γ^{n-1} when o is the observation.

Because a one-step policy choice that optimizes the backed-up value of an information state also corresponds to a vector in Γ^n , it is easy to add individual vectors to Γ^n : simply perform backups for individual information states and add to $\tilde{\Gamma}^n$ the vectors that corresponds to the one-step policy choices that optimize the backed-up values. In fact, because V^n can be represented by a finite set of vectors Γ^n , it must be possible to construct Γ^n by performing backups for a finite number of information states. The problem is to identify a finite set of information states such that performing a backup for all of them, and constructing the corresponding vectors, is guaranteed to generate a set of vectors $\tilde{\Gamma}^n$ that represents V^n (where $\tilde{\Gamma}^n$ may be pruned to yield a minimal set Γ^n). There are various ways of determining a finite set of information states that guarantees this. Algorithms that use this strategy to perform a dynamic-programming update include the “one-pass” algorithm of Smallwood and Sondik [102, 103], the linear-support and relaxed-region algorithms of Cheng [18], and the Witness algorithm of Kaelbling, Littman, Cassandra [50]. Cassandra [11] gives a very clear review of these algorithms.

There is a second strategy for computing the dynamic-programming update that is conceptually simpler and I will review that strategy in the rest of this section. It avoids performing backups for individual information states. Instead, it uses the fact that each one-step policy choice corresponds to a potential vector in Γ^n . The strategy is to generate all potential vectors in $\tilde{\Gamma}^n$ by enumerating all possible one-step policy choices, either explicitly or implicitly, and then to prune dominated vectors to yield a minimal set Γ^n .

The simplest implementation of this strategy consists of two steps. First it exhaustively generates all possible vectors. Then it prunes them to find a minimal

set. There are $|A||\Gamma^{n-1}|^{|O|}$ possible vectors, however, and generating all of them is infeasible when $|O|$ is more than four or five, even if $|\Gamma^{n-1}|$ is relatively small.

A more efficient implementation of this strategy avoids exhaustive generation of all vectors by interleaving generation and pruning. It exploits the fact that each vector $\gamma \in \tilde{\Gamma}^n$ can be expressed as the sum of $|O| + 1$ vectors: an immediate reward vector for taking action a and a vector for each possible observation. Let r^a denote the $|S|$ -vector of one-step rewards for action a . Let $\gamma^{a,o,k}$ denote the $|S|$ -vector associated with a particular action a , observation o , and successor vector $\gamma_k^{n-1} \in \Gamma^{n-1}$, defined as follows:

$$\gamma^{a,o,k}(s) = \beta \sum_{s' \in S} \gamma_k^{n-1}(s') Pr(o|s', a) Pr(s'|s, a), \forall s \in S. \quad (6.5)$$

Given this notation, each potential vector $\gamma_i^n \in \Gamma^n$ can be expressed in vector notation as a sum of vectors, as follows:

$$\gamma_i = r^{\alpha(i)} + \sum_{o \in O} \gamma^{\alpha(i), o, \tau(i, o)}. \quad (6.6)$$

To see how this points to an algorithm for interleaving vector pruning with vector generation, we first introduce some additional notation for operations on sets of vectors. Let \oplus denote the *cross sum* of two sets of vectors, defined as

$$\Gamma \oplus \Gamma' = \{\gamma + \gamma' | \gamma \in \Gamma, \gamma' \in \Gamma'\}, \quad (6.7)$$

where Γ and Γ' are sets of $|S|$ -vectors and $\gamma + \gamma'$ denotes vector addition. This operation is commutative and associative. Therefore, it can be performed on more than two sets of vectors. We introduce the following notation for a cross-sum operation on more than two sets, analogous to the summation operator \sum :

$$\oplus_{i=1}^N \Gamma_i = \Gamma_1 \oplus \Gamma_2 \oplus \dots \oplus \Gamma_N. \quad (6.8)$$

1. *Input:* A set of vectors Γ^{n-1} that represents value function V^{n-1} .
2. For each pair of action a and observation o , generate a set of all potential vectors $\tilde{\Gamma}_{a,o}^n$. Prune each of these sets to yield a minimal set $\Gamma_{a,o}^n$.
3. For each action a :
 - (a) Let Γ_a^n be initially empty.
 - (b) For each observation o , let $\tilde{\Gamma}_a^n = \Gamma_a^n \oplus \Gamma_{a,o}^n$ and prune to yield a minimal set Γ_a^n .
4. Let $\tilde{\Gamma}^n = \cup_{a \in A} \Gamma_a^n$ and prune $\tilde{\Gamma}^n$ to yield a minimal set Γ^n .
5. *Output:* A set of vectors Γ^n that represents value function V^n .

Table 6.3. Incremental pruning algorithm for dynamic-programming update.

Now let

$$\tilde{\Gamma}_{a,o}^n = \left\{ \gamma^{a,o,k} \mid \gamma_k^{n-1} \in \Gamma^{n-1} \right\} \quad (6.9)$$

denote the set of all possible vectors associated with action a and observation o . Given these sets of vectors, we use the cross-sum operation to generate all possible vectors associated with action a , as follows:

$$\tilde{\Gamma}_a^n = \{r^a\} \oplus \left(\oplus_{o \in O} \tilde{\Gamma}_{a,o}^n \right). \quad (6.10)$$

Given these $|A|$ sets of vectors, the set of all possible vectors is:

$$\tilde{\Gamma}^n = \cup_{a \in A} \tilde{\Gamma}_a^n. \quad (6.11)$$

Having generated $\tilde{\Gamma}^n$ in this fashion, we can prune it to yield a minimal set Γ^n .

A more efficient approach than generating all possible vectors before pruning is to prune intermediate sets of vectors before $\tilde{\Gamma}^n$ is generated. The first thing we can do is prune each set $\tilde{\Gamma}_{a,o}^n$ to yield a minimal set $\Gamma_{a,o}^n$, before combining them to create $\tilde{\Gamma}_a^n$. If a vector in $\tilde{\Gamma}_{a,o}^n$ is dominated, there can be no information state for which taking

action a , receiving observation o , and then choosing the strategy represented by this vector is optimal. Therefore, pruning this vector cannot affect the minimal set Γ that is being constructed. Independently, both Jiang [49] and Littman, Cassandra, and Kaelbling [50, 63] noticed that this technique can accelerate dynamic-programming updates.

A second thing we can do is prune each set $\tilde{\Gamma}_a^n$ to yield Γ_a^n before combining them to create $\tilde{\Gamma}^n$. Each vector in these sets corresponds to a one-step policy choice that begins with action a . If one of these vectors is dominated, it means there is no information state for which the one-step policy choice that corresponds to this vector is optimal. Littman, Cassandra, and Kaelbling [50, 63] first used this technique to accelerate dynamic-programming updates.

Zhang and Liu [125] have recently introduced a technique that can accelerate this approach further. Instead of generating the cross-sum of all vector sets in Equation 6.10 before pruning, it interleaves pruning with the cross-sum operation as follows:

$$\Gamma_a^n = \text{prune}(\dots \text{prune}(\text{prune}(\Gamma_{a,o_1}^n \oplus \Gamma_{a,o_2}^n) \oplus \Gamma_{a,o_3}^n) \dots \oplus \Gamma_{a,o_{|O|}}^n) \quad (6.12)$$

An algorithm for performing the dynamic-programming update that uses this technique in combination with the other pruning techniques is also called *incremental pruning*. The algorithm is outlined in Table 6.3. Cassandra *et al.* [15] compare it to other algorithms for performing the dynamic-programming update and report that it is empirically the fastest algorithm. They also describe a *restricted-region* generalization of incremental pruning that often outperforms the basic algorithm. Zhang and Lee [124] describe further refinements of incremental pruning.

1. <i>Input:</i> A set of vectors Γ^n representing value function V^n and a set of vectors Γ^{n-1} representing value function V^{n-1} .
2. residual := 0
3. For each γ in Γ^n
(a) Solve the linear program of Figure 6.1 with inputs γ and Γ^{n-1} .
(b) if (d > residual) then residual := d
4. <i>Output:</i> residual

Table 6.4. Algorithm for computing Bellman residual.

6.4 Value iteration

As in the completely observable case, value iteration for POMDPs consists of iteration of the dynamic-programming update. It can find an optimal value function for a finite-horizon POMDP and can also find an arbitrarily good estimate of the optimal value function for an infinite-horizon POMDP.

For infinite-horizon problems, value iteration converges to an ϵ -optimal value function after a finite number of iterations. As in the completely observable case, the ϵ -optimality of the value function can be detected by computing the Bellman residual. Table 6.4 shows a procedure for computing the Bellman residual under the assumption that the initial value function V^0 is an underestimate of the optimal value function. It is easy to create an initial value function that satisfies this assumption: for example, the value function created by evaluating any finite-state controller satisfies this assumption. If this assumption cannot be made, a more complex algorithm for computing the Bellman residual is described by Littman [63].

A greedy policy with respect to a value function V is defined as follows:

$$\delta(\pi) = \arg \max_{a \in A} \left[r(\pi, a) + \beta \sum_{o \in O} P(o|\pi, a) V(T(\pi|a, o)) \right]. \quad (6.13)$$

If a value function is optimal, a greedy policy with respect to it is optimal. If a value function is ϵ -optimal, a greedy policy with respect to it is ϵ -optimal. If the action associated with each vector γ_i is denoted $act(i)$, a policy can be extracted more simply as follows:

$$\delta(\pi) = act \left(\arg \max_k \sum_{s \in S} \pi(s) \gamma_k(s) \right). \quad (6.14)$$

6.5 Sondik's policy-iteration algorithm

Value iteration solves a POMDP by iteratively improving a value function and extracting a policy from it. A second approach to solving infinite-horizon POMDPs is policy iteration, which iteratively improves a policy. The rest of this chapter reviews a policy-iteration algorithm for POMDPs developed by Sondik [104, 103]. The review begins with a discussion of his choice of policy representation, which significantly influences the algorithm.

Policy representation

For value iteration, it is sufficient to have a representation of the value function because a policy can be defined implicitly by the value function and Equation 6.13. For policy iteration, a policy must be represented independently of the value function because the policy-evaluation step computes the value function of a given policy.

Sondik's choice of policy representation is influenced by Blackwell's proof that, for an infinite-horizon MDP with a continuous state space, a stationary Markov policy is optimal [7]. This means that an optimal policy can be expressed as a stationary mapping from Π to \mathcal{A} . This is the representation of a policy Sondik adopts. It parallels representation of the value function as a stationary mapping from Π to \mathcal{R} . To keep this representation of a policy finite, Sondik further limits policy space by defining an *admissible policy* as a mapping from a finite number of regions of Π to \mathcal{A} . Each region of information space is represented by a finite set of linear inequalities.

where each linear inequality corresponds to a boundary of the region. Thus, a policy induces a partition of Π into a finite number of polyhedral regions that are mapped to actions.

This is Sondik's canonical representation of a policy. However his algorithm makes use of two other representations. In the policy-evaluation step, he converts a policy to an equivalent (or approximately equivalent) finite-state controller that can be more easily evaluated. In the policy-improvement step, he represents a policy implicitly by a value function and Equation 6.14 before converting it back to his explicit representation. Sondik's policy-iteration algorithm is difficult to implement because of the complexity of translating between these three different representations of a policy. A simplified version of his algorithm is outlined in Figure 6.5.

Policy evaluation

Sondik's representation of a policy as a mapping from a finite set of polyhedral regions of information space to action space makes the policy-evaluation step of his algorithm particularly difficult. There is no known method for computing the value function of a policy represented in this way. However, there is a straightforward way to compute the value function of a policy represented as a finite-state controller, as described in Section 2.3. Therefore the policy-evaluation step of Sondik's algorithm first converts a policy from a representation as a mapping from information space to action space to a representation as a finite-state controller. The difficulty of converting between these two representations — often only an approximate conversion is possible — makes Sondik's algorithm difficult to implement.

Sondik links these two representations of a policy by means of the concept of a *finitely transient policy*. A policy is said to be finitely transient if it is a stationary mapping from Π to \mathcal{A} that corresponds to a finite-state controller. More precisely, a policy δ is said to be finitely transient if there is a partition of information space into

<ol style="list-style-type: none"> 1. <i>Input</i>: An initial policy, δ, represented as a mapping from a finite set of polyhedral regions of Π to \mathcal{A}, and a parameter ϵ for detecting convergence to ϵ-optimality. 2. <i>Policy evaluation</i>: Choose $k \geq 1$ as degree of approximation. <ol style="list-style-type: none"> (a) Convert δ to representation as a finite-state controller that is either equivalent or an approximation of degree k. The conversion is accomplished in two steps: <ol style="list-style-type: none"> i. Construct a Markov partition V^k with degree of approximation k. ii. Construct a Markov mapping \hat{v} from V^k. (b) Compute the value function for the finite-state controller by solving the system of equations given by 2.8. 3. <i>Policy improvement</i>: Perform the dynamic-programming update on the concave hull of the value function to create a new value function. The improved policy δ' is implicitly represented by the new value function using Equation 6.14. Convert this implicit representation to an explicit mapping from a finite number of regions of Π to \mathcal{A}. 4. <i>Termination test</i>: If the Bellman residual is $\leq \epsilon(1 - \beta)\beta$, go to step 5. Otherwise set δ to δ' and go to step 2. 5. <i>Output</i>: An ϵ-optimal policy.
--

Table 6.5. Sondik's policy-iteration algorithm.

a finite number of regions such that (i) the action associated with each region is the same as $\delta(\pi)$ for every information state π in the region, and (ii) for any observation, all information states in one region map onto the same region under this policy. Sondik calls such a partitioning of information space a *Markov partition* because the next region is completely determined by the current region, action, and observation. A Markov partition can be used to construct a *Markov mapping* — Sondik's phrase for the transition function of a finite-state controller — where each region in the partition corresponds to a memory state of the controller.

Every admissible policy δ induces a partition of Π into a finite number of regions. The partition of Π induced by a policy δ may not be a Markov partition, however, even if the policy is finitely transient. But, if it is not a Markov partition, there may

still be a *refinement* of the partition that preserves the mapping from Π to \mathcal{A} given by δ and is also a Markov partition. (One partition is said to be a refinement of another if it divides its regions into smaller subregions.) This possibility makes it very difficult to determine whether a policy is finitely transient. This difficulty motivates Sondik's formal definition of a finitely transient policy.

Definition (Sondik) *A finitely transient policy is a stationary policy such that, after some finite period of time, an information state cannot lie at a point in Π at which $\delta(\pi)$ is discontinuous.*

This definition can seem mysterious at first. But it provides a way of detecting whether a policy is finitely transient as well as an algorithm for creating an equivalent (or approximately equivalent) finite-state controller. Consider two information states in the same region of the partition induced by a policy. If, after the same action and observation, they do not map to the same region, then the partition is not Markov. It does not necessarily follow that the policy is not finitely transient because it may be possible to refine this partition further to create a Markov partition that preserves the mapping from Π to \mathcal{A} given by δ . To try to construct such a partition, Sondik relies on the observation that $T(\pi|a, o)$ preserves “straight lines.” This means that two neighboring points in information space, after the same action and the same observation, map to neighboring states in information space. It follows that if two information states in one region do not map into the same region, there must be some information state between them that maps onto a boundary between these regions.

This points to the following algorithm for refining a partition to possibly create a Markov partition. The algorithm starts with the boundaries of a partition, that is, the information states at which δ is discontinuous, denoted D_δ . It applies the inverse mapping $T^{-1}(\pi|a, o)$ to these boundaries to yield a set of potential new boundaries. Potential boundaries that do not lie in regions of Π assigned action a by δ are elim-

inated. Sondik denotes the resulting set of boundary points by D_1 and refines the partition of Π by adding these boundaries. The process is repeated with the new boundaries until no new boundaries can be created.

If there is some finite number k after which this algorithm cannot create new boundaries, the policy must be finitely transient. The number k is called the *degree* of the policy. The idea is that if it is possible to limit the number of steps after which it is impossible to land on a policy discontinuity, it is also possible to limit the number of distinct regions needed to create a Markov partition for a policy and the policy must have an equivalent finite-state controller. If there is no such limit, two information states arbitrarily close together can require different sequences of actions resulting in a value function that is no longer piecewise linear. Therefore, this algorithm provides a way of detecting a finitely transient policy and constructing a Markov partition and corresponding finite-state controller.

However, not all policies are finitely transient. To evaluate policies that are not, Sondik describes a technique for creating an approximate Markov mapping for any stationary policy. A parameter k is chosen that represents the desired degree of approximation and the algorithm described above is applied for k steps. If it does not determine that the policy is finitely transient with degree $\leq k$, it creates an approximate Markov mapping of degree k by choosing an arbitrary point π in each region i , and, for each observation o , sets the approximate Markov mapping $\hat{v}(i, o)$ equal to the region that contains $T(\pi|a, o)$. This mapping \hat{v} is approximate because $T(.|a, o)$ may not map all points in one region to the same next region. But Sondik proves that an approximation can be found that has a value function arbitrarily close to the value function of δ by setting the degree of approximation k high enough; as k increases, the approximation improves.

This is a very high-level summary of Sondik's complex algorithm for policy evaluation. But it conveys some sense of why this step of his algorithm is difficult to

implement. Implementation is easiest for the case $|S| = 2$ because region boundaries can be viewed as points on a line segment and the inverse mapping T^{-1} only needs to be applied to points. For $|S| > 2$, regions boundaries are sections of 1 and applying the inverse operator, and checking the resulting sets of potential boundaries, is a daunting task.

Policy improvement

The policy-improvement step of Sondik's algorithm uses the same dynamic-programming update that value iteration uses. However the dynamic-programming update can only be performed if the value function of the policy being improved is piecewise linear and convex. The value function of a finitely transient policy is always piecewise linear, but it is not necessarily convex.

It is piecewise linear because it is represented by a finite number of vectors, one for each memory state of the controller that has been evaluated. But it is convex only if the controller is started in the memory state that optimizes the value of the starting information state. This is not necessarily true of the controller created by Sondik's technique. It is started in the memory state that corresponds to the region of information space containing the starting information state.

The same controller can represent different policies depending on the rule for selecting the starting memory state. If the controller is started in the memory state that optimizes the value of the starting information state, the policy it represents may not even be a stationary mapping from Π to A . Some vector of the value function computed in the policy-evaluation step may be dominated. If it is, the controller cannot be started in the memory state that corresponds to this memory state although it may eventually enter it. If it does, the same information state may be mapped to two different actions. In this case, as Sondik points out, the controller corresponds to a *semi-Markov policy*.

The important point is that Blackwell's policy improvement theorem is guaranteed to monotonically improve *any* suboptimal policy. Sondik appeals to this to justify performing the policy-improvement step on the policy that has a piecewise linear and convex value function, that is, the (possibly) semi-Markov policy that corresponds to starting the controller in the memory state that optimizes the value of the starting information state. Its value function is the *convex hull* of the value function of the policy evaluated in the policy-evaluation step, which means it is the smallest convex set that contains that value function. This is defined as follows:

$$\bar{V}(\pi|\delta) = \max_j [\pi\gamma_j] \quad (6.15)$$

For every information state, this value function is as good or better than the value function of the policy evaluated in the policy-improvement step, and so the (possibly) semi-Markov policy that corresponds to starting the controller in the memory state that optimizes the value of the starting information state is as good or better than the policy evaluated in the policy-improvement step. If it is not optimal, the policy-improvement step will improve it further. Therefore, Blackwell's policy-improvement theorem applies to Sondik's algorithm.

After the dynamic-programming update is performed, a new stationary policy δ^n is represented implicitly by the updated value function V^n and Equation 6.13 and must be converted to an explicit representation. Each vector γ_i in the improved value function corresponds to a convex polyhedral region V_i of Π that is the solution of a finite system of linear inequalities.

$$V_i = \{\pi \in \Pi : \pi\gamma^i \leq \pi\gamma^j, j = 1, 2, \dots, |\Gamma|\}. \quad (6.16)$$

For each vector $\gamma^j \neq \gamma^i$, there is a potential boundary between regions i and j in the form of a solution to the linear equation, $\pi\gamma^i = \pi\gamma^j$. Usually a region can be defined

by a subset of these boundaries. Extraneous boundaries can be detected and removed by solving a linear program for each potential boundary, as described in appendix B of [102].

Convergence

If a policy is sub-optimal, it will be improved in the policy-improvement step. If it cannot be improved, it must be optimal because the optimality equation is satisfied. This provides a test for optimality and Sondik's policy-iteration algorithm is able to detect convergence to an optimal policy. Because the space of admissible stationary policies is uncountably infinite, however, his policy-iteration algorithm is not guaranteed to converge to an optimal policy after a finite number of iterations. But it is guaranteed to converge to an ϵ -policy after a finite number of iterations. The ϵ -optimality of a policy can be detected by computing the Bellman residual each iteration, as in policy iteration for completely observable MDPs.

Discussion

Sondik's policy-iteration algorithm is almost never used in practice because of the prohibitive difficulty of translating between different policy representations. This is not the only limitation of his algorithm. His method for converting a policy from a representation as a mapping from Π to \mathcal{A} to a representation as a finite-state controller relies on an assumption that limits its applicability. The transformation $T(\pi|a, o)$ must be one-to-one for the algorithm to work because its inverse is used to build a Markov partition and $T^{-1}(\pi|a, o)$ is not a valid function unless $T(\pi|a, o)$ is one-to-one. But, for many POMDPs, the assumption that $T(\pi|a, o)$ is one-to-one is violated. For example, any action that resets memory violates this assumption.

This may explain a curious fact. Sondik mentions using policy iteration to find a solution that is within 10% of optimal for the inspection/maintenance problem of Example 4.1. He reports that his algorithm achieves this result in 6 seconds. A

natural question is: why not run the algorithm longer to obtain an optimal solution? A possible explanation is that this inspection/maintenance problem contains two actions that reset memory and violate the assumption that $T(\pi|a, o)$ is one-to-one. This may make it impossible for the policy-evaluation step to find a finitely transient policy with a value function that is accurate enough to improve the policy further.

There has been little work on policy iteration for POMDPs since Sondik's work. Lee [61] developed a special-case version of Sondik's policy-iteration algorithm for solving a particular three-state POMDP that can be used to model simple inspection/maintenance problems. I am unaware of any other use of policy iteration for solving POMDPs.

CHAPTER 7

AN IMPROVED POLICY-ITERATION ALGORITHM

The previous chapter reviewed value-iteration and policy-iteration algorithms for POMDPs that rely on a piecewise linear and convex representation of the value function. Because of the difficulty of implementing Sondik's policy-iteration algorithm, almost all subsequent work on POMDPs has relied on value iteration.

Sondik's policy-iteration algorithm is complex and difficult to implement because it uses three different representations of a policy and repeatedly translates between them. The contribution of this chapter is to show that policy iteration can be simplified, both conceptually and computationally, by relying on a single representation of a policy as a finite-state controller. This representation makes policy evaluation straightforward. The challenge is to interpret the dynamic-programming update, which is used for policy improvement, as the transformation of a finite-state controller into an improved finite-state controller. This chapter shows that this interpretation is possible and that it leads to an easy-to-implement policy-iteration algorithm that outperforms value iteration in solving infinite-horizon POMDPs.

7.1 Policy iteration in a space of finite-state controllers

Table 7.1 outlines the new policy-iteration algorithm. Its policy-evaluation step is straightforward; it computes the value function of a finite-state controller by solving a system of linear equations. The policy-improvement step uses the dynamic-programming update to transform a policy δ into an improved policy δ' . In both Sondik's algorithm and the algorithm proposed here, the policy that is improved is

represented as a finite-state controller because that is the form in which it is evaluated. Sondik’s algorithm interprets the dynamic-programming update as the transformation of a finite-state controller into an improved policy represented as a mapping from a finite set of polyhedral regions of Π to A . The policy-iteration algorithm described in this chapter interprets the dynamic-programming update as the transformation of a finite-state controller into an improved finite-state controller. This makes it unnecessary to translate between different policy representations.

7.1.1 Value iteration and finite-state controllers

Before describing how to interpret the dynamic-programming update as the transformation of a finite-state controller, it is useful to review a similar, albeit simpler, interpretation of the dynamic-programming update in value iteration. Kaelbling, Littman, and Cassandra [50, 64, 12] point out that for finite-horizon POMDPs, the solution found by value iteration can be represented by a finite-state controller. Because the POMDP has a finite horizon, the controller does not contain loops.

Value iteration, reviewed in Section 6.4, solves a finite-horizon POMDP by computing a non-stationary value function that consists of a separate piecewise linear and convex value function for each horizon. Beginning with a 0-horizon value function, it uses the dynamic-programming update to construct a value function for each horizon n from the $n - 1$ horizon value function. Each vector in the n -horizon value function is associated with a one-step policy choice that includes the selection of an action, and, for each observation, the selection of a successor vector in the $n - 1$ horizon value function. If each of these one-step policy choices is interpreted as a memory state of a controller, the non-stationary value function found by value iteration corresponds to a finite-state controller in which each memory state is associated with a horizon and a vector in the optimal value function for that horizon. Figure 7.1 shows an example for a POMDP with two actions, L and S, and two observations, p and n. The terminal

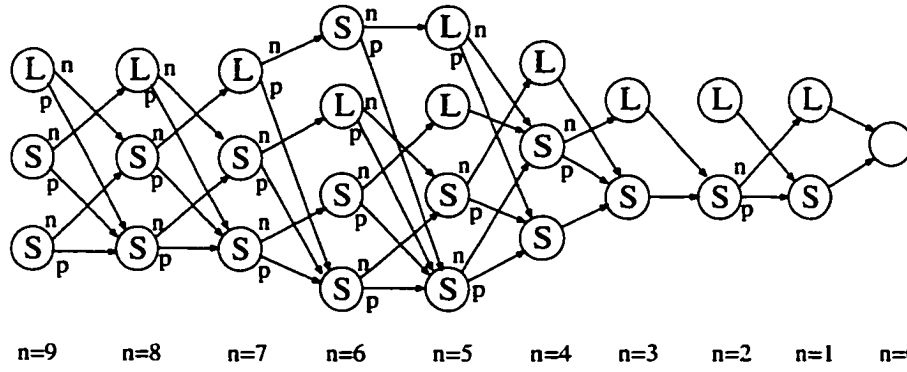


Figure 7.1. Example of an acyclic finite-state controller for a finite-horizon POMDP.

memory state of the controller corresponds to the zero-horizon value function, which is a vector of all zeros that does not have an associated action or successor links. For an n -horizon problem with a starting information state, the controller is started in the memory state that corresponds to the vector in the n -horizon value function that optimizes the value of the starting information state. After the starting information state is used to select the starting memory state, the finite-state controller can be executed without Bayesian updating of an information state.

A finite-state controller for an infinite-horizon problem must include loops to prescribe behavior over an infinite horizon. Kaelbling, Littman and Cassandra note that value iteration for infinite-horizon POMDPs often converges to a piecewise linear and convex value function and, when it does, Cassandra [12] describes a procedure for extracting a cyclic finite-state controller from the vectors in the converged value function. It uses the fact that, when value iteration converges, the $n - 1$ horizon value function is the same as the n -horizon value function (within some degree of approximation — true convergence occurs only in the limit). Because each vector of the n -horizon value function corresponds to a one-step policy choice with links to vectors in the $n - 1$ horizon value function, any link from a vector in the n -horizon value function to a vector in the $n - 1$ horizon value function can be re-directed to the equivalent vector in the n -horizon value function. Redirecting these links cre-

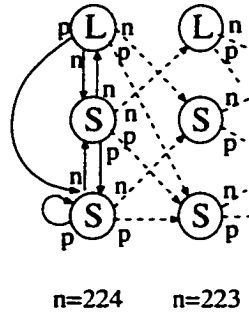


Figure 7.2. Extraction of a cyclic finite-state controller after convergence of value iteration.

ates a cyclic finite-state controller from the value function to which value iteration converges. Figure 7.2 illustrates this for the same example.

This interpretation of the dynamic-programming update as the improvement of a finite-state controller is limited, however, when infinite-horizon POMDPs are considered. A finite-state controller can only be extracted if value iteration converges to an optimal piecewise linear and convex value function, but it does not reliably converge. And if it doesn't (or before it does), it is unclear how to construct a suboptimal finite-state controller from the value function it finds. Nor does this interpretation of the dynamic-programming update support a policy-iteration algorithm. It simply describes how a finite-state controller can be extracted from the optimal value function to which value iteration may converge.

7.1.2 Policy improvement as change of a finite-state controller

The rest of this section describes a generalization of this interpretation of the dynamic-programming update. Instead of simply interpreting it as adding memory states to an acyclic finite-state controller, it interprets the dynamic-programming update as the improvement of a finite-state controller by a combination of three operations. As before, memory states can be added to a controller. But, in addition, memory states can be changed (by changing their associated action and/or successor

1. *Input:* An initial finite-state controller, δ , and a parameter ϵ for detecting convergence to an ϵ -optimal policy.
2. *Policy evaluation:* Compute the value function V^δ for δ by solving the system of equations given by equation (2.8).
3. *Policy improvement:*
 - (a) Perform a dynamic-programming update that transforms a set of vectors Γ^δ representing value function V^δ into a set of vectors Γ' representing an improved value function V' .
 - (b) For each vector γ' in Γ' :
 - i. If the action and successor links associated with it are the same as those of a machine state already in δ , then keep that machine state in δ' .
 - ii. Else if vector γ' pointwise dominates a vector associated with a memory state of δ , *change* the action and successor links of that memory state to those that correspond to γ' . (If it pointwise dominates the vectors of more than one memory state, merge them into a single memory state.)
 - iii. Else *add* a memory state to δ' that has the action and successor links associated with γ' .
 - (c) *Prune* any memory state of δ' for which there is no corresponding vector in Γ' , as long as it is not reachable from a memory state corresponding to a vector in Γ' .
4. *Termination test:* Calculate the Bellman residual and if it is less than or equal to $\epsilon(1 - \beta)/\beta$, go to step 5. Otherwise set δ to δ' . If some node was changed in step 3b, go to step 2; otherwise go to step 3.
5. *Output:* An ϵ -optimal finite-state controller.

Table 7.1. Improved policy-iteration algorithm.

links) and memory states can be pruned. This more general interpretation of the dynamic-programming update supports a policy-iteration algorithm.

The policy-improvement step of the algorithm begins with a dynamic-programming update that determines how the current value function can be improved by adding vectors based on one-step lookahead. Because each vector is associated with a one-step policy choice, it can be viewed as a potential memory state that can be added to

the current finite-state controller — as in the interpretation of value iteration. A comparison of these potential memory states, and their corresponding vectors, with the memory states of the current finite-state controller, and their corresponding vectors, determines how to improve the controller.

First note that because a finite-state controller for an infinite-horizon problem contains loops, it can be in the same memory state at different times. Memory states (and vectors) are no longer associated with a particular horizon. It follows that memory states (and vectors) added by the dynamic-programming update may have duplicates among the memory states (and vectors) of the current finite-state controller (and value function). A memory state is a duplicate of another if it is associated with the same action and, for each observation, has the same successor memory state. (It follows that the corresponding vectors are pointwise equal.) If the dynamic-programming update generates a duplicate memory state (vector), the corresponding memory state in the current finite-state controller is preserved without change. Non-duplicate memory states (vectors) determine how to improve the finite-state controller. The rules for changing the controller are as follows.

- If a non-duplicate vector in Γ' pointwise dominates a vector in Γ^δ , the memory state that corresponds to the pointwise dominated vector in Γ^δ is changed so that its action and successor links match those of the dominating vector in Γ' .
- If a non-duplicate vector in Γ' does not pointwise dominate a vector in Γ^δ , a new memory state is added to the finite-state controller with the same action and successor links used to generate the vector.
- Finally, there may be some memory states in δ for which there is no corresponding vector in Γ' . They can be pruned — but only if they are not reachable from a memory state that corresponds to a vector in Γ' . This last point is important because it preserves the integrity of the finite-state controller.

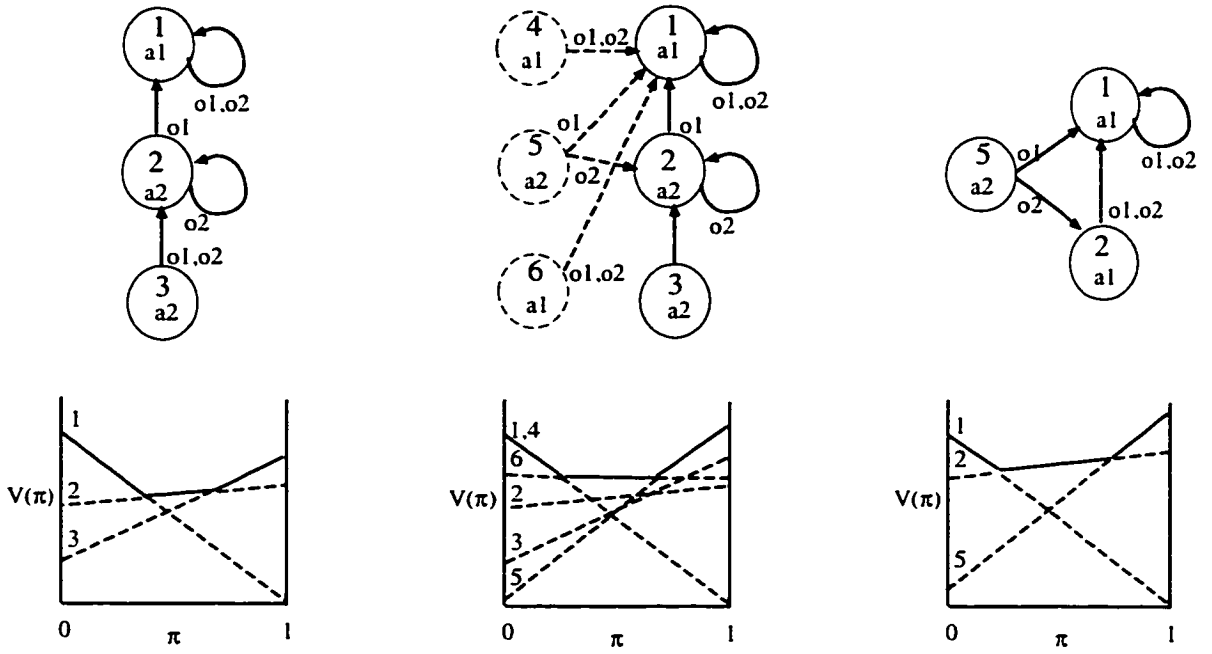


Figure 7.3. Example of policy improvement using dynamic-programming update.

7.1.3 Example

These possible transformations of a finite-state controller are illustrated in Figure 7.3. An initial finite-state controller and its corresponding value function are shown at the left of Figure 7.3. Each memory state of the controller is labeled by a unique number. The corresponding vector in the value function is labeled by the same number. The potential memory states generated by a dynamic-programming update are shown by dashed circles in the middle controller of Figure 7.3. Potential memory state 4 is a duplicate of memory state 1 and causes no change. Potential memory state 5 is not a duplicate and it improves the value function; so it is added to the controller. The vector that corresponds to potential memory state 6 pointwise dominates the vector that corresponds to memory state 2. Therefore, memory state 2 is changed to match the action and successor links of potential memory state 6. Finally, the vector that corresponds to memory state 3 is dominated. Because memory state 3 is not reachable from any other memory state, it is pruned. The improved controller and its corresponding value function are shown at the right of Figure 7.3.

7.1.4 Changing and merging memory states

A policy-iteration algorithm that uses these simple rules to transform a finite-state controller after performing the dynamic-programming update is easy to implement. Moreover, transformation of the finite-state controller after the dynamic-programming update adds little overhead to the policy-improvement step; the running time for policy improvement consists almost entirely of the dynamic-programming update.

Among these three simple transformations, changing memory states is central to policy iteration. In fact, the policy-evaluation step only needs to be performed after a memory state is changed. Changing memory states can improve the value function more than simply adding memory states because part of the policy executed repeatedly within a loop is changed. This allows policy iteration to converge in fewer iterations than value iteration.

Changing a memory state can be viewed as a combination of two transformations: adding a memory state and merging an old memory state into the new one. The idea of merging one memory state into another is an especially useful way of looking at what happens when the vector corresponding to a potential memory state pointwise dominates two (or more) vectors in the current value function. The memory state corresponding to one of these pointwise dominated vectors can be changed to match the potential memory state. But the second pointwise-dominated memory state should be merged into the changed memory state, to simplify the controller. Merging one memory state into another means pruning the memory state and redirecting all links to it to the memory state into which it is merged. We adopt the following rule for merging one memory state into another.

Pointwise dominance test for merging memory states: *If the vector corresponding to one memory state pointwise dominates the vector corresponding to another memory state, the second memory state can be merged into the first.*

7.1.5 Theoretical properties

A proof that the value function of the transformed finite-state controller, δ' , dominates the value function of the previous finite-state controller, δ , can be viewed as a generalization, from scalars to vectors, of Howard's policy-improvement theorem [46]. The key step in the proof is the following lemma. It justifies changing a memory state whenever its vector is pointwise dominated by a vector corresponding to a potential memory state. For this lemma, it is again useful to view changing a memory state as a combination of two transformations; adding a memory state and merging an old memory state into the new one.

Lemma 7.1 *If the vector corresponding to memory state i pointwise dominates the vector corresponding to memory state j , merging memory state j into memory state i will not decrease the value of the controller for any information state.*

Proof: The proof involves reasoning about the effect of behaving as if the two memory states are merged for some number of steps, after which the original policy is followed. The proof is by induction on the number of steps.

Consider behaving so that the first time memory state j is reached, control is passed to memory state i , as if the two memory states had been merged, but every succeeding time, the original policy is followed. Because the vector corresponding to memory state j is pointwise dominated by the vector corresponding to memory state i , this cannot decrease the value of any information state. Now, consider behaving so that the first k times memory state j is reached, control is passed to memory state i , but every succeeding time the original policy is followed. Make the inductive hypothesis that this does not decrease the value of any information state. Then passing control to memory state i the next time memory state j is reached cannot decrease the value of any information state either, by the same reasoning as above. It follows that merging memory state j into memory state i cannot decrease the value of the controller for any information state. \square

Theorem 7.1 *If a finite-state controller is not optimal, policy improvement transforms it into a finite-state controller with a value function that is as good or better for every information state and better for some information state.*

Proof: It is clear that leaving a memory state unchanged or adding a memory state to a finite-state controller cannot decrease its value for any information state. Nor can pruning a dominated memory state, as long as it is not reachable from an undominated memory state. By Lemma 7.1, changing a pointwise-dominated memory state cannot decrease the value of any information state.

It remains to show that if the controller is not optimal, the value of the controller must be improved for at least one information state. If every vector in Γ' is a duplicate of a vector in Γ , the optimality equation is satisfied and the finite-state controller is optimal. If the finite-state controller is not optimal, there must be some non-duplicate vector in Γ' that both changes the finite-state controller and improves the value of some information state. \square

There is a simple test for the optimality of a finite-state controller. If it cannot be improved in the policy-improvement step (i.e., all the vectors in Γ' are duplicates of vectors in Γ), it must be optimal because the value function satisfies the optimality equation. However, this convergence test must be implemented with some care. The optimal value function is unique, but an optimal policy need not be. To prevent oscillation between equivalent finite-state controllers, the policy-improvement algorithm should change a finite-state controller only if doing so improves the value function. This is analogous to the completely observable case.

Policy iteration does not necessarily converge to an optimal finite-state controller after a finite number of iterations because there may not be an optimal finite-state controller. Optimal performance may require infinite memory. Therefore, the same stopping condition used by Sondik can be used to detect ϵ -optimality. Let $\max_{\pi} |V^n(\pi) -$

$V^{n-1}(\pi)$ denote the Bellman residual. Then, the *error bound* of a solution is given by

$$\frac{\beta \max_{\pi} |V^n(\pi) - V^{n-1}(\pi)|}{1 - \beta}. \quad (7.1)$$

A controller is ϵ -optimal when the error bound is less than or equal to ϵ .

Theorem 7.2 *Policy iteration converges to an ϵ -optimal finite-state controller after a finite number of iterations.*

Proof: The proof is a simple extension of the proof that value iteration converges to an ϵ -optimal value function after a finite number of iterations [93]. Each iteration of the dynamic-programming update reduces the error in the value function by a fraction equal to the discount factor and policy evaluation cannot make the value function worse. \square

7.1.6 Efficiency Issues

Apart from improving the efficiency of the dynamic-programming update, there are a couple other ways to improve the efficiency of the policy-iteration algorithm.

7.1.6.1 Policy-improvement preprocessing step

The value function for a finite-state controller has one vector for each of its memory states. Some of these vectors may be dominated, however, which means the finite-state controller cannot be started in the corresponding memory state (although it can eventually enter it). No memory state that has one of these dominated memory states as a successor can be added by the dynamic-programming update. Therefore, the efficiency of the dynamic-programming update can be improved by pruning dominated vectors from Γ^δ before performing the dynamic-programming update. This only affects the input to the dynamic-programming update. The corresponding memory states of the finite-state controller are not pruned, as long as they are reachable from a memory state that corresponds to a vector that is not dominated.

7.1.6.2 Policy evaluation: Decomposition of policy graph

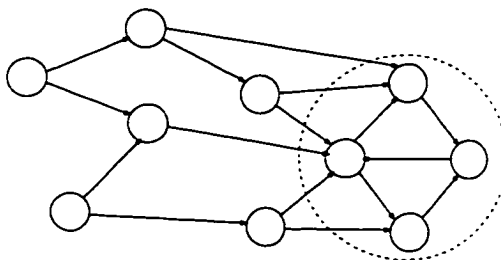


Figure 7.4. Strongly-connected components in policy graph.

The complexity of the policy-evaluation step is $O(|Q|^3|S|^3)$. As the size of the controller grows and/or the size of the state-space grows, policy evaluation can become unreasonably expensive. An effective way to improve its efficiency is to treat the controller as a directed graph and decompose it into strongly-connected components using Tarjan's well-known algorithm [108]. All of the states in the same strongly-connected component are in the same "loop." Figure 7.4 shows a policy graph in which the nodes within the dashed circle are in the same loop, that is, in the same strongly-connected component. Each of the other nodes is in a strongly-connected component by itself. The strongly-connected components can be arranged in a component graph that is acyclic and policy evaluation can be performed separately on each component in backwards order from the leaves of the component graph. Each strongly-connected component can be evaluated as soon as its successor components are evaluated. State values in successor components are treated as constants.

In practice, this often speeds up policy evaluation significantly. The number of memory states in a controller found by policy iteration is often much larger than the number of memory states in its largest strongly-connected component. The controller in Figure 7.4 has ten memory states, but the number of memory states in the largest strongly-connected component is four. Although a contrived example, this is typical of many controllers found by policy iteration. Memory states are first added to "the front" of the finite-state controller and are often not part of any cycle. This method

of policy evaluation is used in the experiments reported in this thesis. In practice, it has reduced the time needed to evaluate large finite-state controllers from several minutes to a few seconds.

7.1.6.3 Modified policy iteration

Although the decomposition technique for policy evaluation described above is effective, the size of the controller and/or the size of the state space can eventually become large enough for exact policy evaluation to be prohibitive. Section 3.2.3 describes a modified policy iteration for completely observable MDPs that replaces exact evaluation of a policy with a more efficient iterative method for estimating the value function. Representation of a policy as a finite-state controller makes modified policy iteration for POMDPs straightforward; the same system of linear equations that is solved exactly can also be solved iteratively by repeatedly using one-step lookahead to re-evaluate the vector that corresponds to each memory state. A modified policy-iteration algorithm may be useful for POMDPs with large controllers and/or large state spaces.

7.2 Performance

To convey a better sense of how the algorithm works, this section describes its performance on a range of examples.

Before describing these examples, there is an issue that affects the implementation of policy iteration (and value iteration) that is important enough to mention. The behavior of both algorithms can be affected by the limited precision of machine arithmetic and the small errors this can introduce into calculations. Both algorithms rely on linear programming to test whether a vector is dominated and often the amount by which a vector is or isn't dominated is close to zero — particularly as the algorithm approaches convergence and improvement of the value function becomes smaller and

smaller. If improvement of the value function, or dominance of a vector, is sufficiently close to zero — as determined by some parameter that reflects the limits of machine precision — it should be treated as if it is zero. Not doing so risks allowing machine imprecision to create spurious memory states, or vectors, that can prevent convergence. Similarly, a parameter that represents the limits of machine precision should be used in the pointwise dominance test of policy improvement. For someone who implements these algorithms, the influence of numerical precision on their behavior cannot be underestimated. Its effect on algorithm performance is discussed further in Sections 7.2.4 and 7.2.5.

The first example considered in this section is the same example used by Sondik to test his policy-iteration algorithm.

7.2.1 Marketing example

Actions	Transition probabilities	Observation probabilities	Expected reward																						
	<table><tr><td></td><td>B</td><td>N</td></tr><tr><td>B</td><td>0.8</td><td>0.2</td></tr><tr><td>N</td><td>0.5</td><td>0.5</td></tr></table>		B	N	B	0.8	0.2	N	0.5	0.5	<table><tr><td></td><td>p</td><td>n</td></tr><tr><td>B</td><td>0.8</td><td>0.2</td></tr><tr><td>N</td><td>0.6</td><td>0.4</td></tr></table>		p	n	B	0.8	0.2	N	0.6	0.4	<table><tr><td>B</td><td>4</td></tr><tr><td>N</td><td>-4</td></tr></table>	B	4	N	-4
	B	N																							
B	0.8	0.2																							
N	0.5	0.5																							
	p	n																							
B	0.8	0.2																							
N	0.6	0.4																							
B	4																								
N	-4																								
Market luxury product (L)																									
	<table><tr><td></td><td>B</td><td>N</td></tr><tr><td>B</td><td>0.5</td><td>0.5</td></tr><tr><td>N</td><td>0.4</td><td>0.6</td></tr></table>		B	N	B	0.5	0.5	N	0.4	0.6	<table><tr><td></td><td>p</td><td>n</td></tr><tr><td>B</td><td>0.9</td><td>0.1</td></tr><tr><td>N</td><td>0.4</td><td>0.6</td></tr></table>		p	n	B	0.9	0.1	N	0.4	0.6	<table><tr><td>B</td><td>0</td></tr><tr><td>N</td><td>-3</td></tr></table>	B	0	N	-3
	B	N																							
B	0.5	0.5																							
N	0.4	0.6																							
	p	n																							
B	0.9	0.1																							
N	0.4	0.6																							
B	0																								
N	-3																								
Market standard product (S)																									

Figure 7.5. Parameters for Example 7.1.

Example 7.1 (Sondik [104]) *A simple two-state, two-action, two-observation POMDP is used to model the problem of finding an optimal marketing strategy given imperfect information about consumer brand preference [104, 103]. The two states of the problem represent consumer preference or lack of preference for the manufacturer's brand; let B denote brand preference and N denote lack of brand preference. Brand preference cannot be directly observed but can be inferred based on purchasing behavior; let p denote purchase of the product and let n denote no purchase. There are*

two marketing actions: the company can market a luxury version of the product (L) or a standard version (S). The luxury version is more expensive to market but has greater potential for profit and increases brand preference. However, consumers are more likely to purchase the less expensive, standard product. The transition probabilities, observation probabilities, and reward function for this example are shown in Figure 7.5. The discount factor is 0.9.

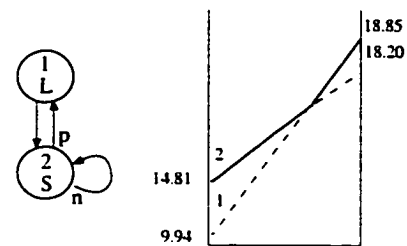
If the algorithm is started with an initial finite-state controller that is equivalent to the initial policy Sondik used, Figure 7.6 shows the transformation of the controller into an optimal finite-state controller. It takes three iterations (and a fraction of a second), the same number of iterations Sondik reports for his algorithm. The top row shows the initial finite-state controller and its corresponding value function. Each memory state is labeled by a unique number, and the vector in the value function that corresponds to it is labeled by the same number. Successive iterations of the algorithm are shown in successive rows of Figure 7.6.

At the beginning of each iteration, the dynamic-programming update is performed. The left-hand side of Figure 7.6 shows the potential memory states it finds by showing what the controller would look like if these memory states are added to the current finite-state controller, as in the value-iteration interpretation of the dynamic-programming update. The potential memory states are indicated by dashed circles.

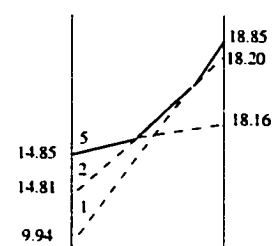
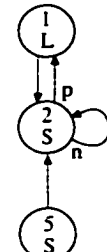
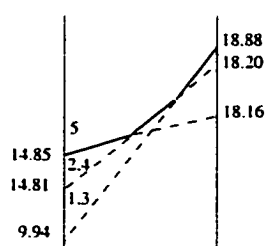
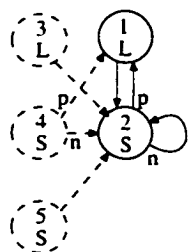
The right-hand side of the figure shows what the controller looks like after applying the policy-improvement procedure to these potential memory states, and their corresponding vectors. In the first iteration, potential memory state 3 is a duplicate of current memory state 1 and potential memory state 4 is a duplicate of current memory state 2. However, memory state 5 is not a duplicate and it is added to the controller.

In the second iteration, the vector for potential memory state 6 pointwise dominates the vector for current memory state 1, the vector for potential memory state

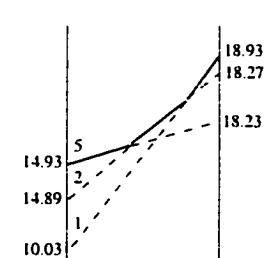
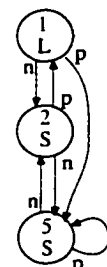
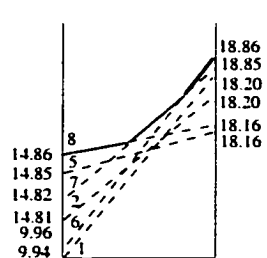
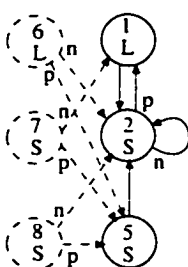
0)



1)



2)



3)

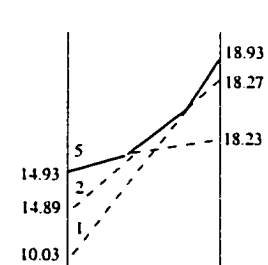
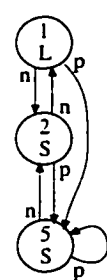
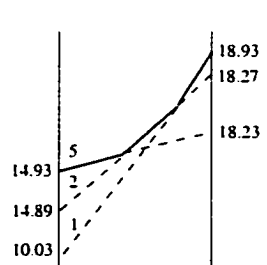
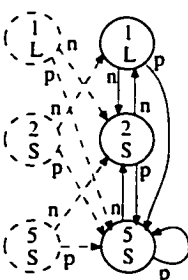


Figure 7.6. Convergence of policy iteration on Example 7.1.

7 pointwise dominates the vector for current memory state 2, and the vector for potential memory state 8 pointwise dominates the vector for current memory state 3. Each memory state is changed accordingly, and the resulting finite-state controller is evaluated.

In the third iteration, each of the three potential memory states is a duplicate of a memory state already in the controller. This signals convergence to an optimal finite-state controller.

For comparison, Figure 7.1 shows the acyclic finite-state controller computed by value iteration for the same marketing problem. Figure 7.1 shows the cyclic finite-state controller extracted from the converged value function, for this example.

This POMDP has an optimal finite-state controller that is very simple. However, none of its actions reset memory and the controller may visit an infinite number of information states. Therefore, it is a POMDP that neither LAO* (chapter 5) nor multistep policy iteration (chapter 4) can solve.

7.2.2 Machine-maintenance example revisited

Section 4.3.1 describes a machine maintenance example, first used as an example by Smallwood and Sondik [102], that can be solved using multistep policy iteration. The same problem can be solved using policy iteration and Figure 7.7 shows the sequence of improving finite-state controllers. The initial controller, shown in the upper left, always chooses the manufacture action. Policy iteration converges to an optimal finite-state controller, shown in the lower right, after 11 iterations and 16 seconds. The eleventh iteration detects that the controller is optimal.

For comparison to the finite-state controller shown in Figure 4.1, Figure 7.7 includes pointers to the memory states that optimize each of the three possible states of perfect information: knowledge that there are 0, 1, or 2 machine malfunctions.

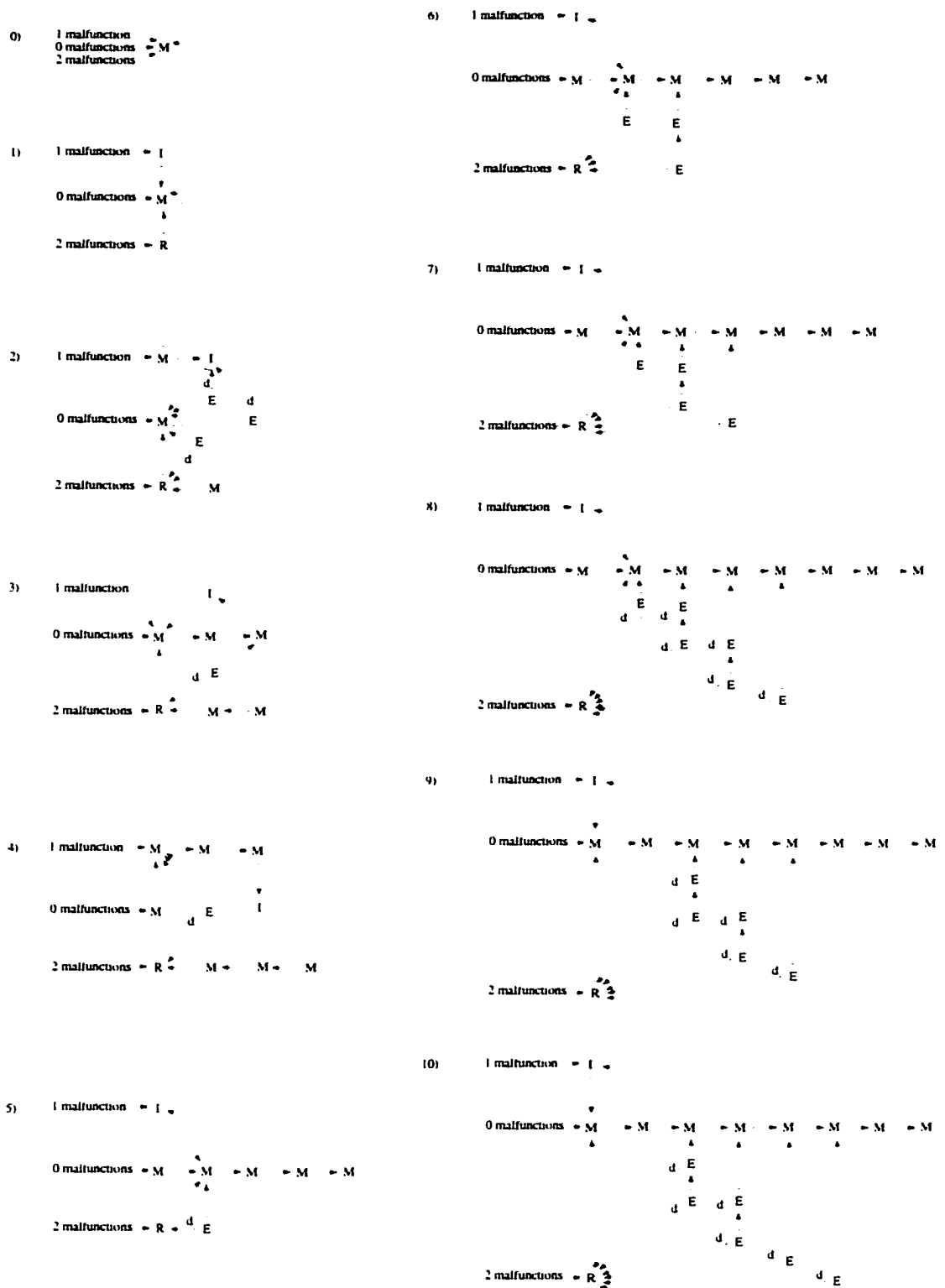


Figure 7.7. Convergence of policy iteration on Example 4.1.

A few differences from the finite-state controller shown in Figure 4.1 are worth mentioning. First, policy iteration finds a finite-state controller that has only one memory state for the inspect action whereas the finite-state controller shown in Figure 4.1 has two, one for each of two different information states. Second, the optimal finite-state controller shown in the lower right-hand corner of Figure 7.7 has memory states that multistep policy iteration does not find. These are present because policy iteration finds a controller that optimizes the value of all possible starting information states; multistep policy iteration only optimizes the values of starting in the states of perfect information. For some information states, it is optimal to start the controller in one of the memory states with an Examine action. But, the controller cannot be in one of these memory states after the second time step and none of these additional memory states can be reached from a memory state of the controller found using multistep policy iteration.

A couple of the memory states in Figure 7.7 are shaded to show that their corresponding vector is dominated by the vectors in the rest of the value function. It is interesting to note that memory states that are dominated after the policy-improvement step, but are retained to preserve the integrity of the controller, may not be dominated after policy evaluation. Conversely, memory states that are not dominated after policy improvement may become dominated after policy evaluation, and are sometimes eligible for pruning; for example, this is true of the shaded memory state in the finite-state controller of iteration 8. Allowing dominated memory states appears essential for this algorithm, but it is unclear how to predict when they will occur.

Section 4.3.1 considered the effect of reducing the cost of the Examine action for this example. Figure 4.2 shows the finite-state controller found by multistep policy iteration for the problem with a reduced Examine cost. However, it is not optimal. The new policy-iteration algorithm can find an optimal finite-state controller and does

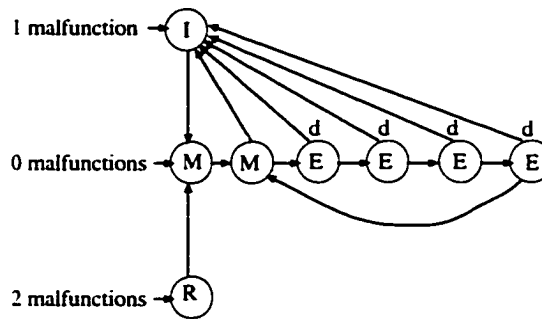


Figure 7.8. Optimal policy for Example 4.1 with changed parameters.

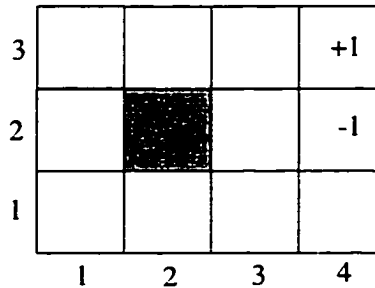


Figure 7.9. Gridworld from Russell and Norvig [97].

so in 18 iterations and 61 seconds. The controller has 36 memory states and most of these optimize different possible starting information states. For comparison to Figure 4.2, Figure 7.8 shows only those memory states of the optimal controller that are reachable from one of the three possible states of perfect information. Notice that the controller contains a cycle that does not include an action that resets memory. That is why multistep policy iteration cannot find this optimal policy.

7.2.3 4×3 gridworld

We have looked at several examples for which policy iteration can find a simple, optimal finite-state controller. Now, we look at how policy iteration behaves on an example for which there does not appear to be a simple, optimal finite-state controller. This example was first used by Russell and Norvig [97] as an illustration of planning under uncertainty. A partially observable version of it is described by Parr and Russell [85].

iteration	cumulative CPU seconds	size of controller	error bound
0	0	1	
1	< 1	3	11.8
2	1	13	10.0
3	5	30	8.2
4	29	135	1.1
5	182	459	0.92
6	1693	1571	0.36
7	40858	6842	0.17

Table 7.2. Performance of policy iteration on Example 7.2.

Example 7.2 (Russell and Norvig [97]; Parr and Russell [85]) Figure 7.9 shows a 4×3 gridworld with coordinates labeled as (x,y) pairs, so that $(1,3)$ is the location in the top left. There are two reward states, $+1$ and -1 . After entering one of these states, the problem ends. (This is an indefinite-horizon problem, without discounting, and both states $+1$ and -1 can be considered connected to a zero-reward absorbing state.)

A robot can move one step at a time in any of the four directions of the compass; north, south, east, and west. Each action succeeds with probability 0.8; with probability 0.1 it moves in a direction that is 90 degrees to one side of its intended direction and with probability 0.1 it moves in a direction that is 90 degrees to the other side of its intended direction. If its movement would take it out of the grid or into a wall, the robot remains in the same state. For example, moving east from location $(1,3)$ will move the robot to location $(2,3)$ with probability 0.8, to location $(1,2)$ with probability 0.1, and the robot will stay in the same location with probability 0.1. A cost of 0.04 is charged for every action.

In this gridworld, partial observability takes the following form. The robot can only see walls to its east and west. Therefore, states $(1,1)$, $(1,3)$, and $(3,2)$ are indistinguishable, as are states $(2,1)$, $(2,3)$, $(3,1)$, and $(3,3)$.

Table 7.2 summarizes the result of running policy iteration on this example. It shows that the size of the finite-state controller grows dramatically each iteration, and, as it does, the running time of the algorithm slows by an equally dramatic amount.

7.2.4 Approximate dynamic-programming updates

The 4×3 gridworld example illustrates a potential difficulty in performing dynamic programming for POMDPs. The size of a finite-state controller (and the number of vectors in its value function) can grow dramatically each iteration — slowing successive iterations of the algorithm. One way to deal with this problem is to modify the dynamic-programming update so that it only generates some subset of all vectors, or potential memory states, that it would normally generate.

The linear program in Figure 6.1 tests whether or not to prune a vector by computing the maximum amount d by which the vector dominates a value function for any information state. In theory, the vector should be pruned if d is non-positive. At the beginning of this section, I noted that a vector should be pruned if this value d is less than some parameter that represents the precision of machine arithmetic, for example, 10^{-10} , and not simply if it is equal to or less than zero. Relaxing the “precision parameter” further, to some value such as 10^{-3} , for example, has the effect of pruning many vectors that improve the value function only marginally. As a result, the dynamic-programming update generates a smaller set of vectors to represent the updated value function. This can be called an “approximate dynamic-programming update.”

Because this has the effect of limiting the number of vectors in the updated value function, as well as the number of memory states in the corresponding finite-state controller, it allows more iterations of dynamic programming to be performed in the same amount of time. Even though the error bound may not be reduced by as much

iteration	cumulative CPU seconds	size of controller	error bound
0	0	1	
1	< 1	3	11.8
2	1	12	10.0
3	5	30	8.2
4	22	86	1.1
5	65	149	0.92
6	132	161	0.65
7	243	219	0.38
8	436	234	0.18
9	784	289	0.065
10	1294	296	0.030
11	1839	324	0.029

Table 7.3. Performance of policy iteration with relaxed precision parameter on Example 7.2.

each iteration, this can speed convergence of the algorithm significantly. Intuitively, one can think of it as focusing computation on improving the most useful vectors, or memory states, by pruning those that are only marginally useful. Table 7.3 shows the performance of policy iteration on the same example tested in the previous section when the precision parameter is relaxed from 10^{-10} to 10^{-3} . By focusing on improving a smaller controller before building a larger one, this approach accelerates convergence of dynamic programming significantly. As the algorithm begins to converge, the precision parameter can be tightened to create further improvement.

Although very useful in practice, this technique is a heuristic and finding the best value of the precision parameter for a particular problem involves trial-and-error. More problematically, its effect on the performance of both policy iteration and value iteration is not perfectly defined or understood. The algorithm can “misbehave” in subtle ways if this parameter is relaxed too much. Many issues about how to perform approximate dynamic-programming updates in this way (or in any other way) need further study. Nevertheless, even a crude approach such as this can be very beneficial. It points to a tradeoff between the value of adding vectors to a value function (or memory states to a controller) to create some improvement of the value function in

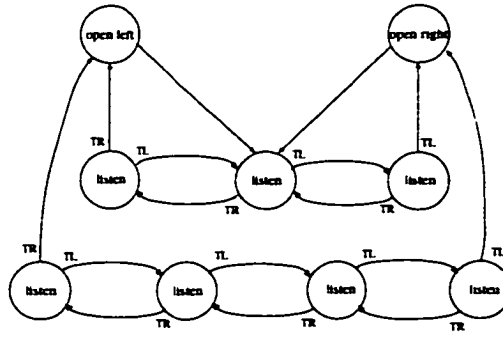


Figure 7.10. Optimal policy for Example 5.1 found by policy iteration.

early iterations of the algorithm, versus the slow-down effect these vectors have on subsequent iterations of the algorithm that might improve the value function further. This tradeoff suggests that the rate at which the value function can be improved may be optimized by adjusting this precision parameter appropriately.

7.2.5 Tiger example revisited

The previous section showed that relaxing a precision parameter can improve the rate of convergence of dynamic programming for some problems by preventing an explosion of vectors (or memory states) from one iteration to the next. For at least one example, relaxing this parameter does more than accelerate convergence. It seems to be necessary for convergence.

Section 5.1 describes a “tiger POMDP” first used as a simple illustration by Cassandra, Kaelbling and Littman [14]. In chapter 5, this problem is solved using LAO*. Figure 7.10 shows the optimal finite-state controller that is found when it is solved using policy iteration. It includes five additional memory states not shown in Figure 5.1. They are present because dynamic programming finds a finite-state controller (and value function) that optimizes all possible starting information states, and not just a particular starting information state. None of these memory states are reachable from the starting information state assumed in Section 5.1.

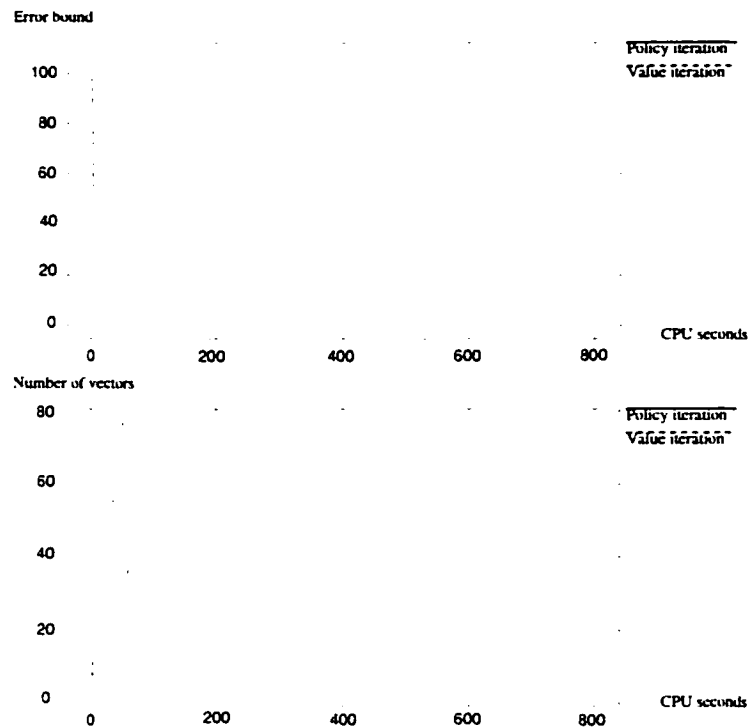


Figure 7.11. Convergence of error bounds (above) and number of vectors (below) for Example 5.1.

Although policy iteration finds a simple, optimal finite-state controller for this example, there is one respect in which it is not a simple problem to solve — possibly because of the complex looping structure that relates these additional memory states. If the precision parameter is set at an amount that reasonably reflects machine precision, say 10^{-10} , the number of vectors (and memory states) grows each iteration and the algorithm does not appear to converge, at least in a reasonable amount of time. If the precision parameter is relaxed to about 10^{-4} , more than can be attributed to machine imprecision but still a small amount, the algorithm converges to the optimal finite-state controller shown in Figure 7.10, although not before first building a more complex controller with more than 70 memory states.

It is interesting that this behavior is not unique to policy iteration; value iteration behaves similarly for this example. Unless the precision parameter is relaxed by the

same amount, the number of vectors in the value function found by value iteration grows unreasonably and value iteration does not converge.

The top panel of Figure 7.11 compares the rate at which the error bound converges using policy iteration and value iteration. For both algorithms, the precision parameter is set to 10^{-4} . Policy iteration converges to an optimal policy after 18 iterations and 61 seconds. Value iteration finds a value function with the same number of vectors as the optimal value function, and an error bound of 0.55, after 71 iterations and 800 seconds, which is as much as this graph shows. After 200 iterations and 1031 seconds, it has reduced the error bound to 0.00077. (Using value iteration, the error bound converges to zero only in the limit.)

The bottom panel of Figure 7.11 illustrates an unusual feature of the performance of both algorithms on this problem, which appears to reflect the difficulty of finding the simple optimal solution. Policy iteration builds a controller with as many as 77 memory states before suddenly converging to the optimal controller, which has only 9 memory states. Value iteration builds a value function with as many as 51 vectors before gradually recognizing that the value function can be represented by only 9 vectors. For each subsequent iteration not shown in this graph, value iteration continues to update a value function composed of 9 vectors.

I am not aware of any other problem for which policy iteration and value iteration have as much difficulty as they do with this one. However, for several other problems (although not for most), relaxing the precision parameter by some lesser amount helps — and may be necessary for — convergence. For such problems, there is also a tendency for the controller, or value function, to grow temporarily larger than the simple solution that is eventually found. Examples for which this is true of policy iteration are also examples for which it is true of value iteration. There is a striking parallel in the behavior of both algorithms, although the reason for it is unclear. The

fact that policy iteration and value iteration behave similarly in this respect indicates that this behavior does not reflect anything inherently problematic for policy iteration.

7.2.6 Comparison to value iteration

A well-known advantage of policy iteration over value iteration is that it can take fewer iterations to converge to ϵ -optimality. Policy iteration improves the value function in two ways each iteration, by the dynamic-programming update and by policy evaluation, whereas value iteration improves the value function in only one of these ways, the dynamic-programming update.

For completely observable MDPs, the fact that policy iteration converges to ϵ -optimality in fewer iterations is not a clear advantage, especially for large MDPs, because the policy-evaluation step is more complex computationally than the dynamic-programming update. For POMDPs, policy evaluation has low-order polynomial complexity compared to the worst-case exponential complexity of the dynamic-programming update [65]. Therefore, policy iteration appears to have a clearer advantage for POMDPs.

Testing bears this out. Figure 7.4 compares the convergence of policy iteration and value iteration on eight small POMDPs. Four of these test problems are described in this thesis; the marketing, maintenance, tiger, and 4×3 gridworld problems. The remaining four are taken from the literature (as are the first four) and are also used as test problems by Cassandra, Littman and Zhang [15]. The cheese grid problem, a simple problem involving a mouse searching in a maze for a piece of cheese, is described by McCallum [78]. It has 11 states, 4 actions and 6 observations. The shuttle problem is a hypothetical space shuttle docking problem first used as an example of a POMDP by Chrisman [19]. It has 8 states, 3 actions, and 5 observations. The network problem is a hypothetical network monitoring problem created by Littman and briefly described in his thesis [64]. It has 7 states, 4 actions, and 2 observations. The

Test problem	Precision parameter	Error bound							
		$\epsilon = 10.0$		$\epsilon = 1.0$		$\epsilon = 0.1$		$\epsilon = 0.01$	
		secs.	iter.	secs.	iter.	secs.	iter.	secs.	iter.
Marketing	10^{-10}	< 1	5	3	27	6	49	9	71
		< 1	3	< 1	3	< 1	4	< 1	5
Maintenance	10^{-10}	289	142	827	372	1363	601	1918	830
		7	7	11	9	13	10	16	11
Cheese grid	10^{-10}	11	7	109	27	319	71	534	116
		11	6	11	6	11	6	11	6
Tiger	10^{-4}	263	18	780	60	861	105	942	150
		3	4	16	7	31	10	43	13
4×3 grid	10^{-4}	2	4	8947	17	59635	62	118112	114
		1	2	862	6	4860	9	10681	12
Shuttle	10^{-6}	17011	30	21659	71	26119	116	30251	160
		75	6	148	7	247	8	369	9
Network	10^{-4}	10448	66	19040	111	27815	156	36496	201
		111	7	784	11	1493	14	2370	18
Aircraft ID	10^{-2}	61472	17	273678	59				
		772	5	11548	7				

Table 7.4. Comparison of how soon policy iteration and value iteration converge to ϵ -optimality on eight small POMDPs.

aircraft identification problem is described in an appendix of Cassandra's thesis [12]. It has 12 states, 6 actions, and 5 observations.

The first four problems — marketing, maintenance, cheese, and tiger — have simple, optimal finite-state controllers. The remaining four do not appear to, and the number of memory states in the finite-state controller that is found (or the number of vectors in the value function) grows into the hundreds, and sometimes the thousands, after a few iterations. For this comparison, basic incremental pruning was used to perform dynamic-programming updates for both algorithms. The initial value function for value iteration was the value function of the initial policy for policy iteration, giving both algorithms the same starting point. The initial policy was always a simple one-state finite-state controller that performs some arbitrary action repeatedly.

Table 7.4 compares the time (in CPU seconds) and the number of iterations each algorithm took to converge to ϵ -optimality, for four different values of ϵ ; 10.0, 1.0, 0.1,

and 0.01. For each test problem, the results for value iteration are shown above the results for policy iteration. For this range of examples (and for all other examples tested by the author), policy iteration is consistently and significantly faster than value iteration.

As explained before, the precision parameter was relaxed to 10^{-4} for the tiger problem to allow convergence to a simple optimal solution. For the four problems without simple optimal solutions, it was also relaxed to prevent the number of vectors in the value function from exploding. This was necessary to allow enough iterations of the algorithm to reduce the error bound to 0.01, although the error bound was not reduced this far for the aircraft identification problem. In every case, the same precision parameter was used for both value iteration and policy iteration.

For the maintenance problem, value iteration takes more iterations to converge than it does for the other problems because the discount factor for the maintenance problem is 0.99, compared to 0.95 for the other problems. The closer the discount factor is to 1.0, the slower value iteration converges. If the discount factor for the other problems is changed to 0.99, value iteration converges much more slowly for them as well. The sensitivity of value iteration to the discount factor is well-known and Sondik appeals to it as a motivation for developing a policy-iteration algorithm for POMDPs. The fact that the convergence rate of policy iteration is not sensitive to the discount factor gives it a further advantage over value iteration.

This comparison measures the rate at which the error bounds for each algorithm converge. A policy based on the value function found by value iteration may be better than its error bound; the same can be true for a policy found by policy iteration. Nevertheless, a comparison of the convergence of error bounds is a reasonable way to compare two dynamic-programming algorithms and, for these examples, there is a clear advantage for policy iteration.

7.3 Change of policy space

The difference between Sondik's policy-iteration algorithm and the policy-iteration algorithm described in this chapter is not simply a difference in choice of policy representation. It is also a difference in the space of possible policies each considers.

The policy-iteration algorithm described in this chapter searches in a policy space of deterministic finite-state controllers. Each finite-state controller has a piecewise linear and convex value function (under the assumption that it is started in the memory state that optimizes the value of an initial information state.)

The policy space searched by Sondik's algorithm includes many policies that do not have piecewise linear and convex value functions. For example, the value function of a finitely transient policy is piecewise linear but not necessarily convex, and the value function of a policy that is not finitely transient may not even be piecewise linear. Including such policies in policy space forces Sondik to resort to a complicated and difficult-to-implement scheme of approximate policy evaluation.

Excluding policies that do not have piecewise linear and convex value functions simplifies policy iteration, in particular, it simplifies policy evaluation, because all excluded policies are either difficult or impossible to evaluate exactly. It also reduces the size of policy space tremendously. The number of excluded policies is uncountably infinite — the cardinality of the policy space searched by Sondik's algorithm — because there is only a countably infinite number of finite-state controllers.

But, surprisingly, a policy space of finite-state controllers is not contained in the policy space searched by Sondik's algorithm. Not every policy that can be represented as a finite-state controller can be represented as a stationary mapping from Π to \mathcal{A} . Recall that when a finite-state controller is evaluated, each memory state corresponds to a distinct vector in the representation of the value function. There is no reason why the vector that corresponds to some memory state cannot be dominated by the other vectors in the value function. A controller cannot be started in a memory

state that does not optimize the value of some starting information state, but it may eventually enter one of these memory states. This possibility means that some information state may be mapped to two different memory states, and possibly two different actions. Therefore, a policy represented by a finite-state controller does not necessarily correspond to a stationary mapping from Π to \mathcal{A} . (Although a finite-state controller may not correspond to a stationary mapping from Π to \mathcal{A} , the mapping from memory states to actions is stationary.) This distinguishes the class of finitely transient policies, which play such an important role in Sondik's algorithm, from the class of policies that can be represented by a finite-state controller. Every finitely transient policy corresponds to a finite-state controller. But not every finite-state controller corresponds to a finitely transient policy.

By Blackwell's proof, a policy cannot be optimal if it does not correspond to a stationary mapping from Π to \mathcal{A} . Nevertheless, such policies play a crucial role in simplifying policy iteration because they make it possible to ensure that an improved policy found by the dynamic-programming update is always a finite-state controller. This makes it possible to adopt a single representation of a policy as a finite-state controller, with all its advantages.

It might seem to be a disadvantage for the algorithm described in this chapter that an optimal policy for an infinite-horizon POMDP cannot always be represented by a finite-state controller, especially since an optimal policy can always be represented as a mapping from Π to \mathcal{A} . However, Sondik defines an admissible policy as a mapping from a *finite* set of polyhedral regions of Π to \mathcal{A} ; the admissibility restriction is necessary to ensure that a policy has a finite representation that can be manipulated by his algorithm. It is unclear, and seems unlikely, that every optimal policy can be represented finitely in this way. Therefore, it is unlikely that every optimal policy is included in the policy space Sondik's algorithm considers.

As for limiting policy space to finite-state controllers, this is no more limiting than representing the value function as piecewise linear and convex — and this representation of the value function is a prerequisite for performing dynamic programming for POMDPs. In fact, it is the close relationship between representation of a policy as a finite-state controller and representation of a value function as piecewise linear and convex that the new policy-iteration algorithm successfully exploits.

7.4 Question of finite convergence

Policy iteration can detect convergence to an optimal finite-state controller. However, an infinite-horizon POMDP does not necessarily have an optimal finite-state controller. That is why we introduced a convergence test for ϵ -optimality and proved that policy iteration converges to an ϵ -optimal finite-state controller after a finite number of iterations. There remains a natural question: if the optimal policy for a POMDP is a finite-state controller, will policy iteration find it after a finite number of iterations? In practice, it often does. Can we prove that it always does?

For completely observable MDPs, the proof that policy iteration converges after a finite number of iterations depends on the fact that the number of possible policies is finite. For POMDPs, the number of possible policies is infinite. Sondik's policy-iteration algorithm can also detect convergence to an optimal policy, but there is no proof that it converges after a finite number of iterations if there is an optimal finite-state controller.

Nevertheless, there are reasons for thinking a finite-convergence proof for the new policy-iteration algorithm may be possible. Given Sondik's representation of a policy, the number of possible policies is uncountably infinite. When a policy is represented as a finite-state controller, the number of possible policies is only countably infinite. This difference means that the set of all possible finite-state controllers can be enumerated

in sequence such that every finite-state controller occurs at some finite point in the sequence.

Based on this fact, it is easy to describe an exhaustive enumeration algorithm that converges in finite time when there is an optimal finite-state controller. The algorithm enumerates all possible finite-state controllers in order of size, performs policy evaluation on each one, and then performs one step of policy improvement to test whether the optimality equation is satisfied. However, this algorithm is hopelessly inefficient.

The advantage of policy iteration is that it enumerates a sequence of improving policies. Effectively, it prunes every policy that does not dominate the current policy. To prove that it converges to an optimal finite-state controller after a finite number of iterations, however, we must prove that it considers an optimal finite-state controller at some finite point in the sequence of improving policies it finds.

For the time being, only empirical evidence suggests that it does, and that evidence is ambiguous and difficult to interpret. Although it often converges, the size of the finite-state controllers it finds can grow and shrink from one iteration to the next of the algorithm. It is equally unclear that value iteration always converges to an optimal piecewise linear and convex value function when there is an optimal finite-state controller. It has been conjectured that it does [50], but it has not been proven. In fact, the convergence behavior of value iteration is difficult to interpret for some of the same reasons the behavior of policy iteration is difficult to interpret. For example, the number of vectors in the improving value functions it finds can grow and shrink from one iteration to the next.

Clarifying the relationship between the behavior of policy iteration and value iteration in solving infinite-horizon problems, and either proving their finite convergence or showing they don't necessarily converge, is an intriguing area for future research. For now, there are many issues about the behavior of both algorithms that remain

unclear. The contribution of this chapter is to show that policy iteration consistently outperforms value iteration without introducing new difficulties.

CHAPTER 8

POLICY IMPROVEMENT USING HEURISTIC SEARCH

This chapter describes an algorithm that is closely related to the policy-iteration algorithm of the previous chapter. The difference is that it does not use the dynamic-programming update to improve a finite-state controller; instead it uses heuristic search.

The motivation for this change of approach is that the dynamic-programming update is the bottleneck of policy iteration (and value iteration). Policy iteration is faster than value iteration because it takes fewer iterations of the dynamic-programming update to converge. But when a single iteration is computationally prohibitive, policy iteration is as impractical as value iteration.

An approximate dynamic-programming update is one attempt to address this difficulty. It only adds vectors, or memory states, that improve the value function by an amount that exceeds some threshold. This limits the number of memory states added to the controller (and vectors added to the value function), which in turn limits the complexity of subsequent iterations of the dynamic-programming update. But it does so by also limiting the degree of possible improvement. Using approximate dynamic-programming updates, policy iteration cannot improve a controller beyond a certain point because it does not add memory states that improve the value function by an amount that falls below the approximation threshold. As a result, it “converges” to a suboptimal controller. The more relaxed the approximation threshold, or precision parameter, the lower the quality of the controller it finds.

A heuristic-search approach to policy improvement also avoids generating all vectors, or memory states, that can improve the value function. As a result, it also finds a smaller controller, and often a much smaller controller. But it does so in a different way that has several advantages. A dynamic-programming update, even an approximate dynamic-programming update, tries to improve a controller for all possible starting information states. A heuristic-search approach to policy improvement only adds memory states that improve the value of a controller for a single starting information state. Limiting the size of the controller in this way does not limit the degree of possible improvement because it does not set a threshold by which an added memory state must improve the value function. The controller can continue to be improved until its value for the starting information state is optimal. Using heuristic search for policy improvement also allows more fine-grained control of policy improvement. It even avoids use of linear programming, which is the most computationally-intensive part of the dynamic-programming update. These advantages allow the algorithm described in this chapter to solve larger POMDPs than policy iteration can, while still using the same piecewise linear and convex representation of a value function.

Using heuristic search for policy improvement illustrates the same approach used successfully in earlier chapters: combine heuristic search with policy iteration. In fact, the algorithm described in this chapter is very similar to the multistep policy-iteration algorithm of chapter 4. Both use multistep lookahead, or heuristic search, for policy improvement. The difference is that the algorithm described in this chapter adopts a piecewise linear and convex representation of the value function and combines multistep lookahead with the more general policy-iteration algorithm for POMDPs developed in chapter 7. As a result, it can solve a larger class of POMDPs.

1. *Input:* An initial finite-state controller, δ , and a parameter ϵ for detecting convergence to an ϵ -optimal policy.
2. *Policy evaluation:* Compute the value function for δ by solving the system of equations given by (2.8).
3. *Policy improvement:*
 - (a) Expand the search tree until either the lower bound of the starting information state is improved, or the error bound on the value of the starting information state is less than or equal to ϵ .
 - (b) If the error bound is less than or equal to ϵ , go to 4. Otherwise continue.
 - (c) Consider every reachable node in the search tree for which the lower bound has been improved. For each of these nodes in backwards order from the leaves to the root:
 - i. If the action and successor links are the same as those of a memory state of δ , then keep that memory state unchanged in δ' .
 - ii. Else compute the vector for this node and if it pointwise dominates the vector for a memory state of δ , *change* the action and successor links of the pointwise dominated memory state to those of this node. (If it pointwise dominates the vectors of more than one memory state, they can be merged into a single memory state.)
 - iii. Else *add* a memory state to δ' that has the same action and successor links as this node.
 - (d) *Prune* any memory state of δ' that is not reachable from the memory state that optimizes the value of the starting information state.
 - (e) Set δ to δ' . If some memory state of the controller has been changed in (3c), go to 2; otherwise go to 3.
4. *Output:* An ϵ -optimal finite-state controller.

Table 8.1. Heuristic-search policy improvement.

8.1 Policy-improvement algorithm

Figure 8.1 outlines the new algorithm. Like policy iteration, it interleaves policy improvement with policy evaluation. Again, policy improvement consists of two steps. First, potential new memory states are identified, although they are identified using heuristic search instead of the dynamic-programming update. Second, the controller is modified using the same transformations that policy iteration uses.

8.1.1 The search algorithm

Potential new memory states are identified by searching forward from the starting information state. The search tree is the same AND/OR tree described in Section 3.3. Each node of the search tree corresponds to an information state and the root node corresponds to the starting information state. Because the implicit search tree is infinite, there are not terminal values to be backed up. However, upper and lower bounds can be computed for information states at the tip nodes of the explicit search tree and these bounds can be backed-up through the tree to the starting information state. Both upper and lower bounds are maintained for each node of the search tree. The best partial solution is determined by selecting the action that maximizes the upper bound at each decision node reachable from the root node.

In addition to both an upper and lower bound function for evaluating the tip nodes of the search tree, the search algorithm uses another heuristic to select which tip node of the best partial policy tree to expand next. Several node-selection heuristics are possible. We use the following: select for expansion the node (and corresponding information state π) for which the value

$$(V^U(\pi) - V(\pi)) * Pr(\pi|\pi_0) * \beta^{depth(\pi)} \quad (8.1)$$

is greatest, where V^U denotes the upper-bound function, the lower-bound function is the value function V of the current finite-state controller, $Pr(\pi|\pi_0)$ denotes the probability of reaching information state π beginning from the starting information state π_0 at the root of the tree, and $depth(\pi)$ denotes the depth of information state π in the search tree (measured by the number of actions taken along a path from the root). This selection heuristic focuses computational effort where it is most likely to improve the bounds of the starting information state. It also ensures that the search tree will be expanded in such a way that no unpruned search path is ignored

indefinitely. This is important because it ensures that an improved solution, if one exists, will eventually be found.

Because we are only concerned with optimizing the value of the starting information state, the error bound of a solution is the difference between the lower and upper bound values of the starting information state. The test for convergence to an ϵ -optimal policy is an error bound that falls below ϵ . The error bound can be reduced by improving the upper-bound or the lower-bound value of the starting information state. When the lower-bound value of the starting information state is improved, it signals that the finite-state controller can be improved and control is passed from the search algorithm to a procedure for improving the finite-state controller.

8.1.2 Improvement of the finite-state controller

The heuristic-search algorithm I have just described has been used before to solve infinite-horizon POMDPs approximately [98, 60, 113]. For infinite-horizon problems, it can only search to a finite depth and previous algorithms find a solution that takes the form of a tree; the solution tree grows with the depth of the search, but it is always a partial solution to an infinite-horizon problem. The innovation of this chapter is to show how to use heuristic search to find finite-state controllers that include loops.

The key idea is to use the piecewise linear and convex value function of a finite-state controller as a lower-bound function for the search algorithm. When an improved lower-bound value for the starting information state is backed-up through the search tree, it signals that the finite-state controller that corresponds to the lower-bound function can also be improved. The transformations used to improve the controller are the same transformations used by policy iteration. Thus, using the piecewise linear and convex value function of a finite-state controller as a lower-bound function forges a link between a policy-iteration and a heuristic-search approach to solving POMDPs.

Using the value function of a finite-state controller as the lower-bound function means that each node of the search tree is not only associated with an information state, it is also associated with a memory state of the controller. When the search algorithm opens a node of the search tree, it computes its lower-bound value by finding the vector in the value function of the current finite-state controller that optimizes the value of this information state. This establishes a correspondence between nodes of the search tree and memory states of the finite-state controller.

Expanding an OR node (and all its child AND nodes), and backing up its lower bound, is equivalent to performing a one-step backup for the corresponding information state (as in equation 6.3). This backup may improve the lower bound of the information state and, if it does, it signals that a memory state (and corresponding vector) can be added to the finite-state controller that improves the value of at least this one information state. Therefore expanding a search node performs a similar function as the dynamic-programming update, and can be interpreted in a similar way as the (potential) modification of a finite-state controller. The difference is that a node expansion corresponds to a one-step backup for a single information state whereas the dynamic-programming update performs a one-step backup for all possible information states. This gives the heuristic-search algorithm more fine-grained control of policy improvement.

When the lower-bound value of an information state in the search tree is improved, it is backed-up through the search tree and possibly improves the lower-bound value of the starting information state at the root of the tree. When the lower-bound value of the starting information state is improved, the search algorithm has also found a way to improve the finite-state controller. Beginning at the root of the search tree, and selecting the action that optimizes the lower-bound value of each OR node, each reachable node of the search tree for which the lower-bound value has been improved is identified. For each of these nodes, in backwards order towards the root:

- A check is made for whether it is a duplicate of some memory state of the current controller. If it is a duplicate, no change is made. If it is not a duplicate, a vector is computed for this potential memory state based on the one-step policy choice it represents and the vectors that correspond to its successor nodes. (Recall that every node in the search tree corresponds to a vector that is used to compute its lower bound value.)
- If the new vector pointwise dominates a vector associated with some memory state of the current controller, that memory state can be changed to match the action and observation links of the improved node.
- Otherwise a new memory state is added to the controller.

These steps are performed for each reachable node for which the lower bound value has been improved. After all transformations of the controller have been made, any memory state that is not reachable from the memory state that optimizes the value of the starting information state can be pruned without affecting the value of the starting information state.

Finally policy evaluation is invoked to recompute the value function. This is only necessary if some memory state has been changed, not if memory states have simply been added. The error bound of the solution is the difference between the upper and lower bound values for the starting information state.

8.1.3 Example

Figure 8.1 illustrates the algorithm with a simple example. An initial finite-state controller and its value function are shown at the left. There is a pointer to the memory state that optimizes the value of the starting information state, π_0 .

In the middle of Figure 8.1, the dashed nodes and arcs added to the controller represent potential memory states corresponding to a path through the search tree, beginning at the root, for which the lower-bound value of each node has been improved.

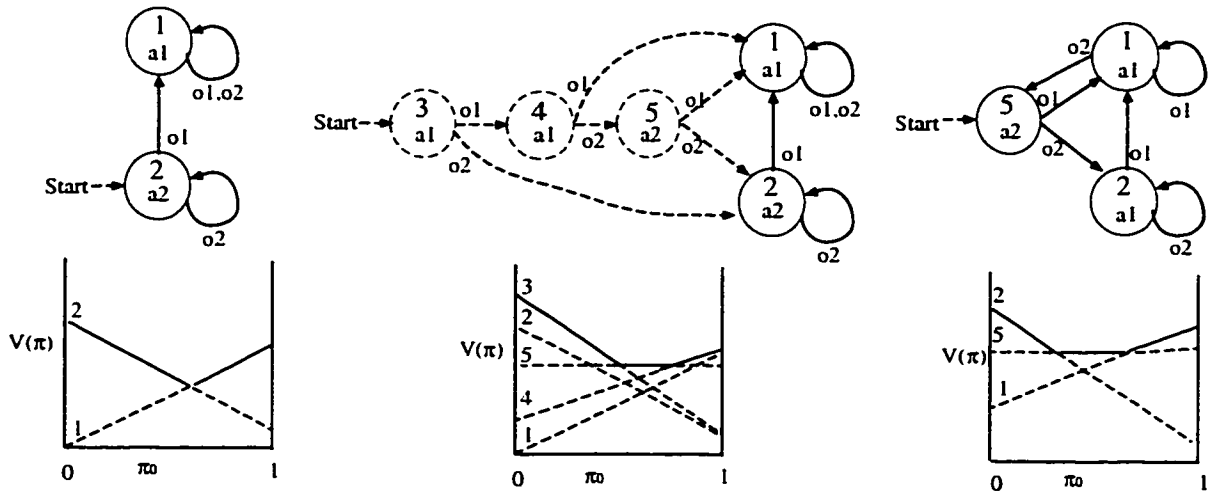


Figure 8.1. Example of policy improvement using heuristic search.

The vector computed for each potential memory state is shown in the associated value function.

Based on these potential memory states, the finite-state controller is transformed as follows. First, potential memory state 5 is added to the controller. Second, memory state 1 is changed so that its action and observation links match those of potential memory state 4. This is done because the vector associated with potential memory state 4 pointwise dominates the vector associated with memory state 1. Finally, potential memory state 3 is a duplicate of memory state 2 and causes no change to the controller. The improved finite-state controller is shown at the right of Figure 8.1, with its corresponding value function.

8.1.4 Theoretical properties

The theoretical properties of this algorithm are similar to those of policy iteration, but are specialized to a starting information state.

Theorem 8.1 *If a finite-state controller does not optimize the value of the starting information state, heuristic-search policy improvement can transform it into a finite-state controller with an improved value for the starting information state.*

Proof: If a finite-state controller does not optimize the value of a starting information state, there must be some sequence of actions and observations in the search tree that improves the value of the starting information state. Because the search tree is expanded in such a way that no unpruned search path is ignored indefinitely, an improving sequence of actions and observations must eventually be found.

Transformation of the current finite-state controller must improve the value of the controller for the starting information state, by the following reasoning. Adding memory states to the controller corresponding to the nodes along this search path for which lower bounds have been improved must improve the value of the controller for the starting information state, and cannot decrease its value for any other information state. By Lemma 7.1, changing memory states of the finite-state controller that are pointwise dominated by any potential memory state along this search path cannot decrease the value of the controller for any information state. Finally, pruning memory states that are unreachable from the starting memory state cannot decrease the value of the controller for the starting information state. \square

Theorem 8.2 *Policy iteration using heuristic search for policy improvement converges after a finite number of steps to a finite-state controller that is ϵ -optimal for the starting information state.*

Proof: The error bound is the difference between the backed-up lower bound and the backed-up upper bound values of the starting information state. The lower bound corresponds to a policy tree in which the action that maximizes the lower bound value is selected at each decision node reachable from the root node. The upper bound corresponds to a policy tree in which the action that maximizes the upper bound value is selected at each decision node reachable from the root node. The difference between the upper and lower bound values of the starting information state is bounded by the total difference between the upper and lower bound values

of the tip nodes of these policy trees, because the reward function for non-tip nodes is the same for both. With expansion of the search tree, these policy trees grow and the contribution of the bound values of their tip nodes to the bound values of the starting information state grows exponentially smaller because of discounting. Their contribution can be made arbitrarily small by expanding the search tree far enough. Therefore the error bound can be made arbitrarily small by expanding the search tree far enough. \square

8.2 Performance

There are two ways to compare the performance of this algorithm to the policy-iteration algorithm of the previous chapter. One is to compare how quickly each algorithm improves the value of a finite-state controller for some starting information state. The other is to compare how quickly each algorithm improves the error bound of the solution.

For all of the eight test problems of Figure 7.4, and for many other problems tested by the author, the heuristic-search algorithm consistently improves the value of the starting information state as quickly, and usually more quickly, than policy iteration. Sometimes, the difference is dramatic. For four of the eight test problems (cheese grid, 4x3 grid, shuttle and aircraft ID), it also improves the error bound more quickly. For the other four, it does not. In fact, sometimes the error bound converges very slowly and remains poor. This can usually be attributed to an upper-bound function that significantly overestimates the optimal value function.

Two of the test problems serve as representative examples; the 4×3 grid problem and the network problem. To compare the performance of both algorithms, the starting information state is assumed to be a uniform probability distribution over system states. The upper-bound function used to evaluate the tip nodes of the search tree is the optimal value function assuming complete observability. This is a simple

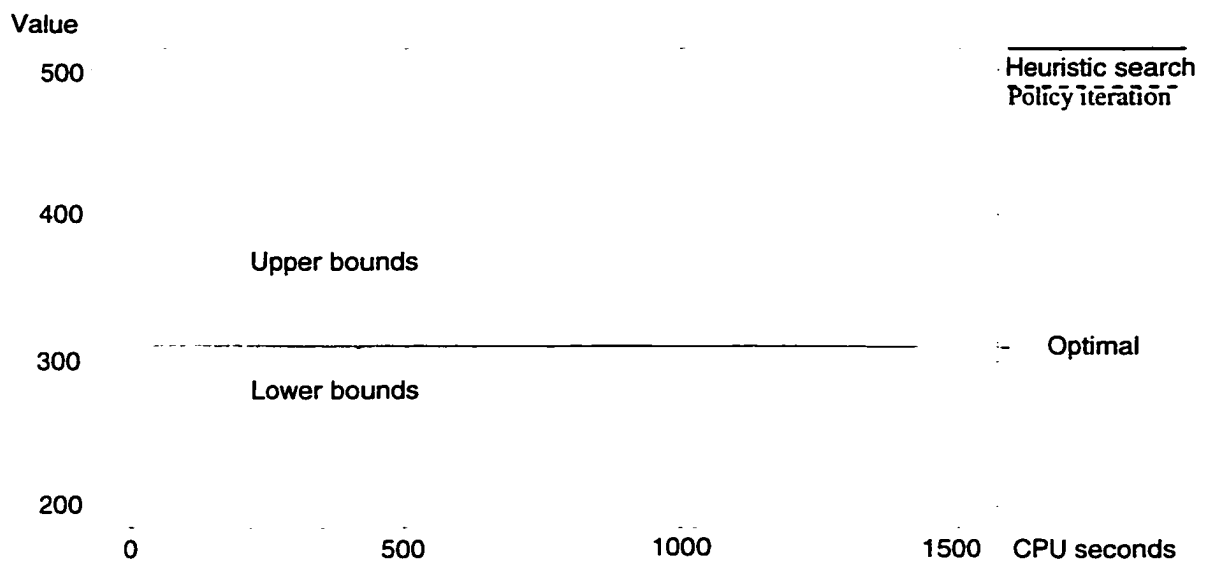


Figure 8.2. Convergence of bounds for 4×3 grid problem using heuristic search and policy iteration.

and widely-used upper-bound function for POMDPs, and was also used in chapters 4 and 5. It is easily computed using a dynamic-programming algorithm for completely observable MDPs.

For the 4×3 grid problem, Figure 8.2 shows that both the lower and upper bound values for the starting information state converge more quickly using heuristic search than using the dynamic-programming update to improve a policy. Heuristic search finds a controller with 52 memory states that optimizes the value of the starting information state and it reduces the error bound to zero in 299 CPU seconds. Figure 8.2 compares its performance to a policy-iteration algorithm that uses approximate dynamic-programming updates with a precision parameter of 10^{-3} ; this precision parameter was chosen because it led to the fastest rate of improvement of policy iteration for this problem. Nevertheless, policy iteration converges more slowly and finds a larger controller. Although the controller it eventually finds has the same value for the starting information state as the controller found by heuristic search, it has 230 memory states; the additional memory states optimize other possible starting information states. Using approximate dynamic-programming updates means the er-

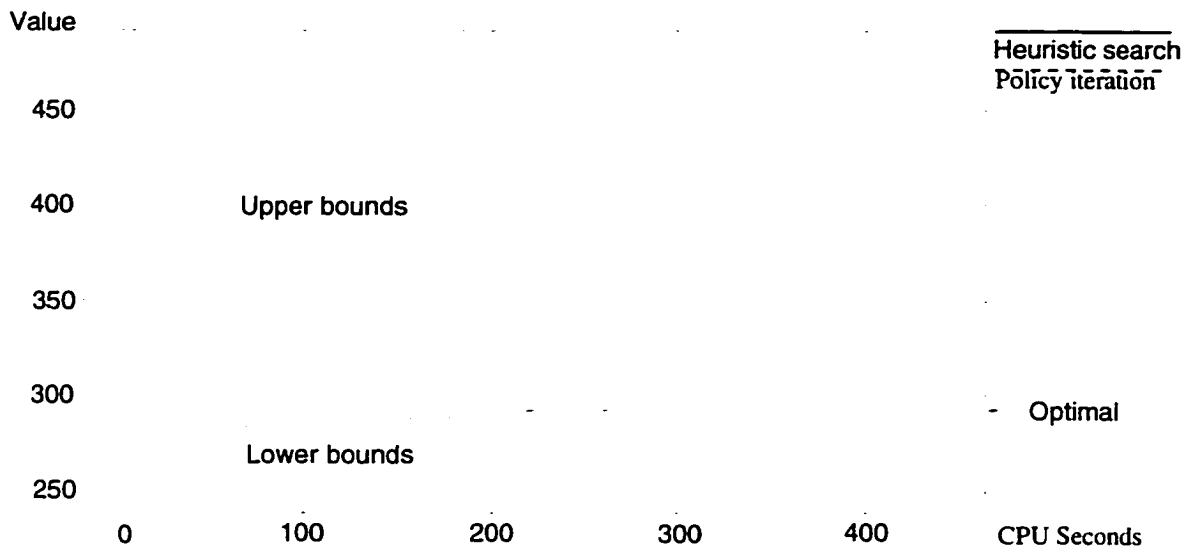


Figure 8.3. Convergence of bounds for network problem using heuristic search and policy iteration.

ror bound is not reduced to zero. Tightening the precision parameter can reduce the error bound further, but it does so by adding more memory states. Figure 7.2 shows that policy iteration with exact dynamic-programming updates builds a controller with thousands of memory states without reducing the error bound to zero.

For the network problem, Figure 8.3 shows that heuristic search improves the lower bound about as quickly as policy iteration. (Again, the comparison is to policy iteration using approximate dynamic-programming updates; this time the precision parameter is 10^{-4} .) It does so by finding a smaller controller. Heuristic search finds a controller with only 25 memory states that performs as well, for the same starting information state, as a controller with 450 memory states found by policy iteration.

For this problem, however, policy iteration improves the error bound more quickly than heuristic search and the error bound of the solution found by heuristic search remains poor. An explanation for this is that the optimal value function assuming complete observability is a poor upper-bound function for this problem and, as a result, the error bound can only be improved by deep expansion of the search tree. Because the quality of the upper-bound function also determines how aggressively

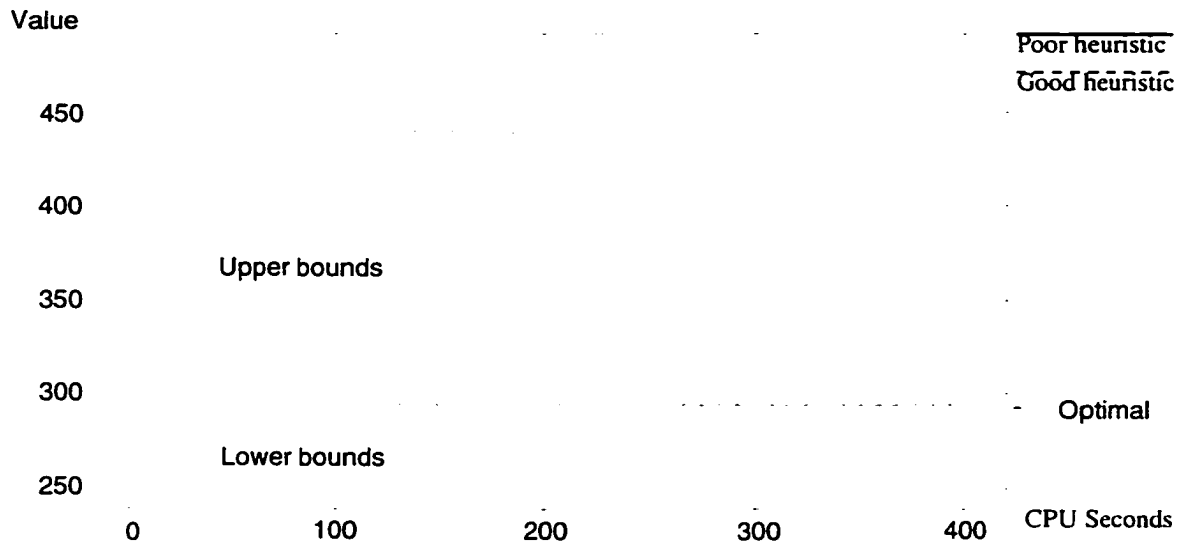


Figure 8.4. Convergence of bounds for network problem using two different upper-bound functions.

the search tree is pruned, improvement of the error bound is slow and the size of the search tree quickly exceeds available memory. Heuristic-search policy improvement reduces the error bound to only 46.98 before running out of memory after 438 seconds.

The importance of a good upper-bound function for reducing the error bound of a solution can be illustrated by comparing the performance of heuristic-search policy improvement using this rather poor upper-bound function with its performance using a much better upper-bound function. A better upper-bound function can be computed by performing a few iterations of policy iteration and then adding the error bound of the solution policy iteration finds to the piecewise linear and convex value function. This simple method for computing an upper-bound function, used in tandem with heuristic search, creates a hybrid approach to solving POMDPs that exploits the strength of both algorithms.

Figure 8.4 compares the performance of heuristic search using these two upper-bound functions. With the better upper-bound function, the heuristic-search algorithm for policy improvement finds a much tighter error bound for the solution. It

also improves the lower-bound value of the starting information state, or equivalently, the finite-state controller, more quickly, although the difference is not as dramatic.

Whether the error bound is reduced more quickly using heuristic search or the dynamic-programming update appears to be problem-dependent. The time it takes heuristic search to reduce the error bound depends on how aggressively the search tree can be pruned. For policy iteration, the time it takes to reduce the error bound depends on how large a controller it generates. For some problems, the error bound can be reduced to zero by finding a small, optimal controller. For other problems, a controller with thousands of memory states may be built without significantly reducing the error bound.

8.3 Discussion

This chapter describes an approach to coping with the inefficiency of the dynamic-programming update for POMDPs. It replaces it with a heuristic-search algorithm that adds memory states to a finite-state controller selectively, and only if they improve the value of a starting information state. This has the following advantages; it eliminates the need to perform the dynamic-programming update (as well as the need to use linear programming), it improves a finite-state controller in an incremental fashion that allows fine-grained control of problem solving, and it focuses computation where it is most likely to improve the value of the starting information state. Because the finite-state controller it finds optimizes the value of a starting information state only, and not the value of every possible information state, it is usually smaller than a controller found by dynamic programming.

An initial implementation of this algorithm performs well. However, there is room for improvement of both the implementation and the algorithm, and the potential of this approach is not fully developed in this chapter. One issue that I have not yet drawn attention to is the cost of policy evaluation. Its complexity, $O(|Q|^3|S|^3)$, can

become a bottleneck as the size of the state space and/or controller grows. For policy iteration, the worst-case exponential complexity of the dynamic-programming update is a more serious bottleneck. When heuristic search is used for policy improvement, the policy-evaluation bottleneck becomes more apparent. This is progress of a sort, however, and there are many techniques for performing policy evaluation approximately, developed for solving large completely observable MDPs, that may be usefully applied here.

Another issue is that the controller found by this search algorithm may not be the smallest of all controllers with the same value for the starting information state. Finding the smallest controller among all controllers with the same value for the starting information state is not a conventional automata-minimization problem. Controllers with the same value for a starting information state may not be equivalent in other respects; in particular, they may not have the same value for other starting information states. But for the purpose of optimizing the value of a starting information state, they are equally useful and it is advantageous to find the smallest one. This is particularly important because the complexity of both the search and the policy-evaluation steps increases with the size of the controller.

The problem of minimizing the size of a controller without decreasing its value for a starting information state does not appear to have an obvious solution. I cannot describe an efficient (i.e., non-exhaustive) algorithm for determining a minimal-size controller, in this sense, or even give a useful criterion for identifying one. (Even when policy iteration converges to an optimal finite-state controller, the part of the controller that is reachable from a particular memory state is not necessarily a minimal-size controller for that starting information state.) However I have developed some useful heuristic techniques for reducing the size of a controller. More work is needed to refine them and I do not describe them in this thesis. They were not used to enhance the performance of the search algorithm for the results reported earlier. However,

for some problems, they can enhance its performance significantly. For example, I reported that the search algorithm found a controller with 52 memory states that optimized the value of the starting information state for the 4×3 grid problem. Using these simplification techniques, a controller with only 33 memory states was found with the same optimal value, and it was found more quickly. These simplification techniques are not as effective for other problems, but other techniques might be. The possibility of simplifying a controller, without decreasing its value for a starting information state, is an important area for further work.

An interesting aspect of the algorithm described in this chapter is the way it relates different approaches to solving POMDPs. Like policy iteration, it represents a policy as a finite-state controller and adopts a piecewise linear and convex representation of the value function. It uses the same policy-evaluation step and the same transformations of the finite-state controller that policy iteration uses. But, instead of using the computationally-intensive dynamic-programming update, it improves a finite-state controller using a heuristic-search algorithm that focuses computation on improving the value of the controller for a starting information state. The efficiency of this heuristic-search approach depends on the quality of the search heuristic and a heuristic evaluation function can be computed by solving a relaxed or approximate version of the problem in which some constraints have been removed to make the problem easier to solve.

CHAPTER 9

CONCLUSION

9.1 Summary of contributions

This thesis describes four new algorithms for solving infinite-horizon POMDPs. All of these algorithms represent a policy as a finite-state controller and exploit this representation to improve the efficiency of problem solving. One advantage of this representation is that it makes policy evaluation straightforward, which is a crucial step of policy iteration. Another advantage is that it makes analysis of reachability easier. This supports a heuristic-search approach that can find a policy that optimizes the value of a starting information state, and not one that optimizes all possible information states. There is little precedent for using heuristic search to solve infinite-horizon problems. It requires generalizing heuristic search to find solutions with loops and this generalization represents an important contribution of this thesis. Showing that heuristic search can be used to solve infinite-horizon MDPs also helps to elucidate the relationship between dynamic programming and heuristic search.

Specific contributions of this thesis include the following:

- A multistep policy-iteration algorithm that uses heuristic search to perform dynamic-programming backups. It can be used to solve any POMDP that includes at least one action that resets memory. It is especially effective for solving problems for which observations provide perfect information, but incur a cost.
- A non-admissible search heuristic that guarantees convergence of multistep dynamic programming to an optimal policy.

- A generalization of heuristic search, called LAO*, that can find solutions with loops. Although more promising as an approach to solving completely observable MDPs, it can also find optimal policies for some POMDPs.
- An interpretation of the dynamic-programming update for POMDPs as the transformation of a finite-state controller into an improved finite-state controller. This supports a policy-iteration algorithm that is easier to implement than an earlier policy-iteration algorithm of Sondik. The new algorithm also significantly outperforms value iteration in solving infinite-horizon POMDPs.
- A heuristic-search approach to policy improvement that avoids some of the difficulties of the dynamic-programming update. Because it focuses computation on regions of information space that are likely to be visited from a starting information state, it usually finds a smaller finite-state controller than policy iteration, consisting only of relevant memory states.

The four algorithms described in this thesis suggest the following classification of POMDPs, based on the class of optimal policies each algorithm can find. This classification provides a convenient way of comparing the algorithms. Each class of POMDPs is a subset of the next.

1. an optimal policy is memoryless (primarily includes completely observable MDPs)
2. an optimal policy is a finite-state controller that resets memory at finite intervals (solved by multistep policy iteration)
3. an optimal policy, starting from a particular information state, is a finite-state controller that visits a finite number of information states (solved by LAO*)
4. an optimal policy, starting from a particular information state, is a finite-state controller (solved by heuristic search for POMDPs)

5. an optimal policy, starting from any information state, is a finite-state controller (solved by policy iteration)
6. an optimal policy requires infinite memory

9.2 Future extensions

POMDPs present a daunting computational challenge and many issues remain to be addressed before this model is practical for realistic-sized problems. An important contribution of this thesis is the framework it establishes for future work. Below I suggest several possible research directions. Many of these, including a memory-bounded approach to approximation, only become feasible once a policy is represented as a finite-state controller. For others, the framework developed in this thesis suggests a fresh perspective.

Improved search heuristics

Three of the four algorithms described in this thesis rely on heuristic search and their efficiency can be dramatically affected by the heuristic that guides the search. This thesis does not directly address the question of how to design good search heuristics for POMDPs, with two exceptions. Chapter 4 introduces a non-admissible search heuristic that guarantees convergence of multistep policy iteration to an optimal policy. And chapter 8 briefly describes adding the error bound to a suboptimal value function computed by policy iteration to create a piecewise linear and convex upper bound function. Otherwise, very simple search heuristics are used. No one factor is likely to improve the performance of the algorithms presented in this thesis more than better search heuristics.

There has been work on value function approximation that is relevant for designing admissible search heuristics. One approach that has been explored is to compute an approximate value function for a POMDP by computing values for a finite grid of

information states and using interpolation to compute the values of information states between the grid points. Because the optimal value function for a POMDP is convex, linear interpolation can compute state values that are upper bounds on optimal state values, providing an admissible heuristic. An optimal value function for a completely observable MDP, used as a search heuristic, illustrates this approach in its simplest form. It can be viewed as a grid that is defined only for states of perfect information. Better search heuristics can be computed by defining the grid for states of imperfect information. Early work used a fixed grid of information states [51, 28, 71]. More recent work has achieved better performance using a variable grid [38, 10].

Another approach to designing search heuristics for POMDPs is to solve a relaxed version of a problem, that is, a simplified version of the problem in which some constraints on the solution have been removed to make it easier to solve. An optimal value function for the simplified problem can be used as an admissible heuristic for the original problem. Again, an optimal value function for a completely observable MDP serves as a simple example. The assumption that observations provide perfect information, even when they don't, creates a relaxed version of the problem that is easier to solve. A more sophisticated development of this idea is an approximation algorithm described by Zhang and Liu [125] that assumes observations reliably reveal the current region of state space. An optimal value function for this simplified POMDP is also an upper bound on the optimal value function for the original POMDP. It seems possible to extend the idea of solving a relaxed version of a POMDP in other directions.

The idea that computing a search heuristic involves solving a POMDP approximately — either by solving a relaxed version of a problem or by approximating the optimal value function in other ways — provides a useful perspective because it relates exact and approximation algorithms for solving POMDPs to each other.

Approximation algorithms can be used to compute a search heuristic that improves the performance of an exact algorithm.

Average-reward performance

A single performance criterion is considered in this thesis: expected discounted reward over an infinite horizon. Another important performance criterion for infinite-horizon problems is average reward per time step. It not only seems possible to extend the results of this thesis to this performance criterion, the framework developed in this thesis may make the average-reward criterion easier to use. Solving average-reward MDPs requires analyzing the cyclic structure of a policy by decomposing it into “recurrent classes,” that is, classes of states within the same loop, because different recurrent classes may have a different average reward. Representing a policy as a finite-state controller can make this analysis easier. Although the literature on solving average-reward POMDPs is sparse [103, 90, 91, 30], Sondik extends his policy-iteration algorithm to average-reward POMDPs. It is likely that the policy-iteration approach developed in this thesis can be extended to average-reward POMDPs as well.

Memory-bounded control

The framework developed in this thesis suggests an approach to approximation that bounds the number of memory states in a controller in exchange for computational speedup. This raises the following question; for a given POMDP, is it possible to find the best finite-state controller with no more than n memory states, for any n ? For a POMDP with a starting information state, this question is well-defined because every finite-state controller can be compared based on its value for the starting information state.

The question is not answered in this thesis, but a positive answer would have important implications. For example, it would settle the question of finite convergence

raised in Section 7.4. An algorithm that can find the best controller with no more than n memory states can be easily modified to find an optimal finite-state controller, if one exists, by increasing n gradually and finding the best controller for each memory bound n until a controller is found that satisfies the optimality equation.

A positive answer to this question would also provide an elegant approximation algorithm that allows a tradeoff between the number of memory states in a controller and the quality of the policy it represents.

Stochastic policies

This thesis considers finite-state controllers that are deterministic. It is possible to define finite-state controllers that are stochastic. A stochastic controller can map the same memory state to different actions, with a probability associated with each action, and can allow the transition from one memory state to the next to be a stochastic function of the observation received.

Stochastic finite-state controllers have been studied extensively in other contexts [88]. For memory-bounded control, in particular, they offer intriguing advantages. For simple hypothesis-testing problems (including so-called “bandit problems”), it has been shown that for any stochastic finite-state controller with n memory states, there is a deterministic finite-state controller with $n + m$ memory states (for some $m < \infty$) that can perform as well or better. But if memory is limited to n memory states, as in a memory-bounded approach, a stochastic finite-state controller can perform at least as well, and arbitrarily better, than the best deterministic finite-state controller with the same number of memory states [42, 43]. In other words, randomization of a policy can improve performance without adding memory.

A stochastic finite-state controller also has a piecewise linear and convex value function that can be computed in a straightforward way by solving a system of linear equations. This suggests extending the approach developed in this thesis to find

stochastic finite-state controllers. Particularly relevant work includes a paper by Platzman that describes a linear-programming approach to iteratively improving a stochastic finite-state controller for an infinite-horizon POMDP [92]. Singh *et al.* [100] describe the advantages of stochastic memoryless policies.

Adaptive control and learning

This thesis considers planning problems for which there is uncertainty about action effects and about the current state. The model of the problem, including the state transition probabilities, observations probabilities, and reward function, is assumed to be known with certainty. This is sometimes an unrealistic assumption. In an *adaptive control* problem, a controller is faced with the additional problem of estimating model parameters. This problem can also be formalized as a POMDP and it may be interesting to use the approach developed in this thesis to attack such problems. Bandit problems and hypothesis-testing problems are well-known examples of adaptive control problems for which there is an interesting body of work on finite-memory approaches [123].

In addition to more traditional work on adaptive control, there has been some recent work on learning a POMDP model by adapting techniques for learning hidden Markov models [54, 99]. This reflects one approach to coping with model uncertainty; learn or estimate the POMDP model and then apply a planning algorithm to the model to solve the problem. A second approach, illustrated by reinforcement learning, is to bypass model learning and try to directly learn a correct policy. There has been some interesting work on reinforcement learning for POMDPs, including learning memoryless policies [69, 100, 48] and finite-length memory policies [79]. However, reinforcement learning has not yet been used to learn general finite-state controllers. The insights in this thesis may help to develop a more general reinforcement-learning approach to POMDPs.

Relationship to other paradigms in AI planning

Problems with large state sets, action sets, or observation sets can be modeled in a more natural and concise manner using structured (that is, compositional or factored) representations [8]. These representations can be exponentially smaller than a representation that explicitly enumerates all possible states, transition and observation probabilities, and rewards. Boutilier and Poole [9] have shown how to adapt some dynamic-programming algorithms for POMDPs to exploit structured representations. Integrating this approach with the finite-memory framework developed in this thesis is a promising approach to solving larger POMDPs.

There has been recent interest in adapting the semi-Markov decision process model to allow hierarchical planning using macro-actions [107, 86], although so far this work has focused on the completely observable case. For POMDPs, representation of a policy as a finite-state controller provides a natural framework for hierarchical control and the results of this thesis may provide some insight about how to extend work on hierarchical control of MDPs to the partially observable case.

Both hierarchical control and use of structured representations are characteristic of AI planning, as are the heuristic-search techniques studied in this thesis. There is much to be gained by integrating research on planning using MDPs with more traditional AI frameworks for planning. Some recent work on planning in a space of probability distributions [57] and extensions of this model to allow observations that provide imperfect information [26, 25] bring traditional AI planning formalisms closer to the POMDP model. Some AI researchers have used finite-state controllers as a plan representation [95, 111]. As research in planning under uncertainty advances, there is reason to expect continued convergence between the interests and insights of researchers who have adopted the MDP model, and are trying to exploit problem structure to solve larger problems, and researchers using traditional AI planning frameworks, who are trying to extend these frameworks to handle uncertainty.

APPENDIX

COMPUTATIONAL ENVIRONMENT

The computational results reported in this thesis were obtained on an AlphaStation 200/4 with a 233Mhz processor and 128M of RAM. Code was written in C and executed under a UNIX operating system. The value-iteration and policy-iteration algorithms compared in chapter 7 use a linear-programming subroutine. For this, the commercial linear-programming package CPLEX was used.

BIBLIOGRAPHY

- [1] Trends and controversies: AI planning systems in the real world. *IEEE Expert* 11 (1996), 4–12.
- [2] Astrom, K.J. Optimal control of Markov decision processes with incomplete state estimation. *Journal of Mathematical Analysis and Applications* 10 (1965), 174–205.
- [3] Astrom, K.J. Optimal control of Markov processes with incomplete state information, II. *Journal of Mathematical Analysis and Applications* 26 (1969), 403–406.
- [4] Barto, A.G., Bradtke, S.J., and Singh, S.P. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72 (1995), 81–138.
- [5] Bellman, R. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.
- [6] Bertsekas, D. *Dynamic Programming and Optimal Control, Vols. I and II*. Athena Scientific, Belmont, MA, 1995.
- [7] Blackwell, D. Discounted dynamic programming. *Annals of Mathematical Statistics* 36 (1965), 226–235.
- [8] Boutilier, C., Dean, T., and Hanks, S. Planning under uncertainty: Structural assumptions and computational leverage. In *Proceedings of the Third European Workshop on Planning (EWSP'95)* (Assisi, Italy, 1995).
- [9] Boutilier, C., and Poole, D. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)* (1996).
- [10] Brafman, R.I. A heuristic variable grid solution method for POMDPs. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)* (Providence, RI, 1997), pp. 727–733.
- [11] Cassandra, A.R. Optimal policies for partially observable Markov decision processes. Tech. Rep. CS-94-14, Brown University, 1994.
- [12] Cassandra, A.R. *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, Brown University, 1998.

- [13] Cassandra, A.R., Kaelbling, L.P., and Kurien, J.A. Acting under uncertainty: Discrete Bayesian models for mobile robot navigation. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS-96)* (1996), pp. 963–972.
- [14] Cassandra, A.R., Kaelbling, L.P., and Littman, M.L. Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)* (Seattle, WA, 1994), pp. 1023–1028.
- [15] Cassandra, A.R., Littman, M.L., and Zhang, N.L. Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-97)* (Providence, RI, 1997), pp. 54–61.
- [16] Chakrabarti, P.P., Ghose, S., Acharya, A., and DeSarkar, S.C. Heuristic search in restricted memory. *Artificial Intelligence* 41 (1989), 197–221.
- [17] Chakrabarti, P.P., Ghose, S., and DeSarkar, S.C. Admissibility of AO* when heuristics overestimate. *Artificial Intelligence* 34 (1988), 97–113.
- [18] Cheng, H. *Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, University of British Columbia, 1988.
- [19] Chrisman, L. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)* (San Jose, California, 1992), pp. 183–188.
- [20] Darrell, T., and Pentland, A. Active gesture recognition using partially observable Markov decision processes. In *Proceedings of the Thirteenth IEEE International Conference on Pattern Recognition (ICPR '96)* (Vienna, Austria, 1996).
- [21] Dean, T., Kaelbling, L.P., Kirman, J., and Nicholson, A. Planning under time constraints in stochastic domains. *Artificial Intelligence* 76 (1995), 35–74.
- [22] Dean, T., and Wellman, M. *Planning and Control*. Morgan Kaufman Publishers, San Mateo, CA, 1991.
- [23] Dearden, R., and Boutilier, C. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence* 89 (1997), 219–283.
- [24] Drake, A.W. *Observation of a Markov Process Through a Noisy Channel*. PhD thesis, Massachusetts Institute of Technology, 1962.
- [25] Draper, D., Hanks, S., and Weld, D. A probabilistic model of action for least-commitment planning with information-gathering. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence (UAI-94)* (Seattle, WA, 1994), pp. 178–186.

- [26] Draper, D., Hanks, S., and Weld, D. Probabilistic planning with information-gathering and contingent execution. In *Artificial Intelligence Planning Systems: Proceedings of the Second International Conference (AIPS-94)* (Chicago, IL, 1994).
- [27] Eagle, J.N. The optimal search for a moving target when the search path is constrained. *Operations Research* 32 (1984), 1107–1115.
- [28] Eckles, J. *Optimum Replacement of Stochastically Failing Systems*. PhD thesis, Stanford University, 1966.
- [29] Ellis, H., Jiang, M., and Corotis, R.B. Inspection, maintenance, and repair with partial observability. *Journal of Infrastructure Systems* 1 (1995), 92–99.
- [30] Fernandez-Gaucherand, E., Arapostathis, A., and Marcus, S.I. On the average cost optimality equation and the structure of optimal policies for partially observable Markov decision processes. *Annals of Operations Research* 29 (1991), 471–512.
- [31] Forbes, J., Huang, T., Kanazawa, K., and Russell, S. The BATmobile: Towards a Bayesian automated taxi. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)* (Montreal, Canada, 1995), Morgan Kaufmann, pp. 1878–1885.
- [32] Haddaway, P., and Hanks, S. Utility models for goal-directed, decision-theoretic planners. *Computational Intelligence* 14 (1998).
- [33] Hansen, E.A. Cost-effective sensing during plan execution. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)* (Seattle, WA, 1994).
- [34] Hansen, E.A. Markov decision processes with observation costs. Tech. Rep. CS-97-01, University of Massachusetts, 1997.
- [35] Hansen, E.A. An improved policy iteration algorithm for partially observable MDPs. In *Advances in Neural Information Processing Systems 10: Proceedings of the 1997 Conference* (Denver, CO, 1998), MIT Press.
- [36] Hansen, E.A., Barto, A.G., and Zilberstein, S. Reinforcement learning for mixed open-loop and closed-loop control. In *Advances in Neural Information Processing Systems 9: Proceedings of the 1996 Conference* (Denver, CO, 1997), MIT Press, pp. 1026–1032.
- [37] Hansen, E.A., and Zilberstein, S. Heuristic search in cyclic AND/OR graphs. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)* (Madison, WI, 1998).

- [38] Hauskrecht, M. Incremental methods for computing bounds in partially observable Markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)* (Providence, RI, 1997), pp. 734–739.
- [39] Hauskrecht, M. *Planning and Control in Stochastic Domains with Imperfect Information*. PhD thesis, Massachusetts Institute of Technology, 1997.
- [40] Hauskrecht, M., and Fraser, H. Planning medical therapy using partially observable Markov decision processes. In *Proceedings of the Ninth International Workshop on Principles of Diagnosis (DX-98)* (Cape Cod, MA, 1998).
- [41] Hauskrecht, M., Merleau, N., Boutilier, C., Kaelbling, L.P., and Dean, T. Hierarchical solution of Markov decision processes using macro-actions. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence* (Madison, Wisconsin, 1998).
- [42] Hellman, M.E. The effects of randomization on finite-memory decision schemes. *IEEE Transactions on Information Theory IT-18* (1972), 499–502.
- [43] Hellman, M.E., and Cover, T.M. On memory saved by randomization. *The Annals of Mathematical Statistics* 42 (1971), 1075–1078.
- [44] Hertzberg, J. On executing crisp plans in uncertain environments. Unpublished manuscript, 1997.
- [45] Horvitz, E.J., Breese, J.S., and Henrion, M. Decision theory in expert systems and artificial intelligence. *Journal of Approximate Reasoning* 2 (1988), 247–302.
- [46] Howard, R.A. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.
- [47] Hughes, J.S. Optimal internal audit timing. *The Accounting Review* 52 (1977), 56–68.
- [48] Jaakkola, T., Singh, S.P., and Jordan, M.I. Monte-Carlo reinforcement learning in non-Markovian decision problems. In *Advances in Neural Information Processing Systems 7: Proceedings of the 1994 Conference* (Denver, CO, 1995), MIT Press.
- [49] Jiang, M. *Partially Observable Markov Decision Process Models for Structural Management Policies and Design*. PhD thesis, John Hopkins University, 1994.
- [50] Kaelbling, L.P., Littman, M.L., and Cassandra, A.R. Planning and acting in partially observable stochastic domains. *Artificial Intelligence* 101 (1998), 99–134.
- [51] Kakalik, J.S. Optimum policies for partially observable markov systems. Tech. Rep. 18, Massachusetts Institute of Technology, 1965.

- [52] Kander, Z. Inspection policies for deteriorating equipment characterized by N quality levels. *Naval Research Logistics Quarterly* 25 (1978), 243–255.
- [53] Klein, M. Inspection–maintenance–replacement schedules under Markovian deterioration. *Management Science* 9 (1962), 25–32.
- [54] Koenig, S., and Simmons, R.G. Unsupervised learning of probabilistic models for robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)* (1996), pp. 2301–2308.
- [55] Koenig, S., and Simmons, R.G. Xavier: A robot navigation architecture based on partially observable Markov decision process models. In *Artificial Intelligence Based Mobile Robotics: Case Studies of Successful Robot Systems*, D. Kortenkamp, R.P. Bonasso, and R. Murphy, Eds. MIT Press, Cambridge, MA, 1997.
- [56] Korf, R.E. Real-time heuristic search. *Artificial Intelligence* 42 (1990), 189–211.
- [57] Kushmerick, N., Hanks, S., and Weld, D. An algorithm for probabilistic planning. *Artificial Intelligence* 76 (1995), 239–286.
- [58] Lane, D.E. A partially observable model of decision making by fishermen. *Operations Research* 37 (1989), 240–254.
- [59] Lark III, J.W. *Applications of Best-First Heuristic Search to Finite-Horizon Partially Observed Markov Decision Processes*. PhD thesis, University of Virginia, 1990.
- [60] Larsen, J.B. *A Decision Tree Approach to Maintaining a Deteriorating Physical System*. PhD thesis, University of Texas at Austin, 1989.
- [61] Lee, C.H. *Optimal Control Limit Policy for a Partially Observable Markov Decision Process Model*. PhD thesis, Texas A&M University, 1994.
- [62] Littman, M.L. Memoryless policies: Theoretical limitations and practical results. In *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior* (1994), MIT Press.
- [63] Littman, M.L. The witness algorithm: Solving partially observable Markov decision processes. Tech. Rep. CS-94-40, Brown University, 1994.
- [64] Littman, M.L. *Algorithms for Sequential Decision Making*. PhD thesis, Brown University, 1996.
- [65] Littman, M.L., Cassandra, A.R., and Kaelbling, L.P. Efficient dynamic-programming updates in partially observable Markov decision processes. Tech. Rep. CS-95-19, Brown University, 1995.

- [66] Littman, M.L., Cassandra, A.R., and Kaelbling, L.P. Learning policies for partially observable environments: Scaling up. In *Proceedings of the Twelfth International Conference on Machine Learning* (San Francisco, CA, 1995), Morgan Kaufman, pp. 362–370.
- [67] Littman, M.L., Dean, T.L., and Kaelbling, L.P. On the complexity of solving Markov decision problems. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence* (1995), AAAI Press.
- [68] Littman, M.L., Goldsmith, J., and Mundhenk, M. The computational complexity of probabilistic plan existence and evaluation. *Journal of Artificial Intelligence Research* (1998).
- [69] Loch, J., and Singh, S. Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes. In *Proceedings of the Fifteenth International Conference on Machine Learning* (1998).
- [70] Lovejoy, W.S. Some monotonicity results for partially observed Markov decision processes. *Operations Research* 35 (1987), 736–743.
- [71] Lovejoy, W.S. A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research* 28 (1991), 47–66.
- [72] Luss, H. Maintenance policies when deterioration can be observed by inspections. *Operations Research* 24 (1976), 359–366.
- [73] MacQueen, J. A test for suboptimal actions in Markov decision problems. *Operations Research* 15 (1967), 559–561.
- [74] Mahedevan, S., Khaleeli, N., and Marchalleck, N. Designing agent controllers using discrete-event Markov models. In *Proceedings of the AAAI Fall Symposium on Model-Directed Autonomous Systems* (Cambridge, MA, 1997).
- [75] Martelli, A., and Montanari, U. Additive AND/OR graphs. In *Proceedings of the Third International Joint Conference on Artificial Intelligence* (1973), pp. 1–11.
- [76] Martelli, A., and Montanari, U. Optimizing decision trees through heuristically guided search. *Communications of the ACM* 21 (1978), 1025–1039.
- [77] McCallum, A.K. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, 1995.
- [78] McCallum, R.A. Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Machine Learning Conference* (Amherst, MA, 1993), Morgan Kaufmann, pp. 190–196.
- [79] McCallum, R.A. Hidden state and reinforcement learning with instance-based state identification. *IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics* 26 (1996), 464–474.

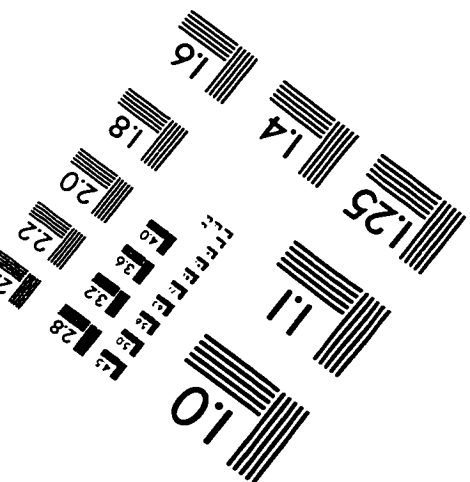
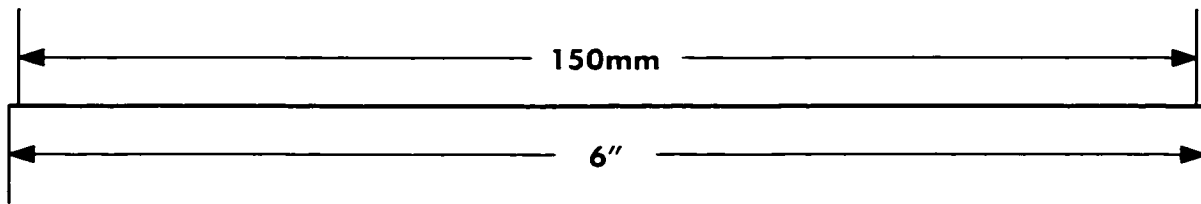
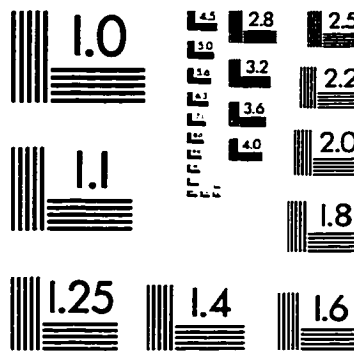
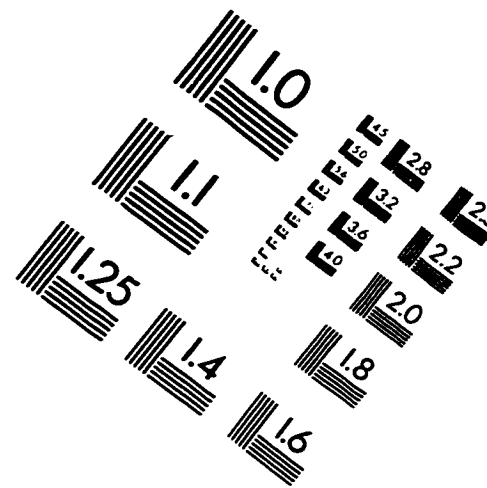
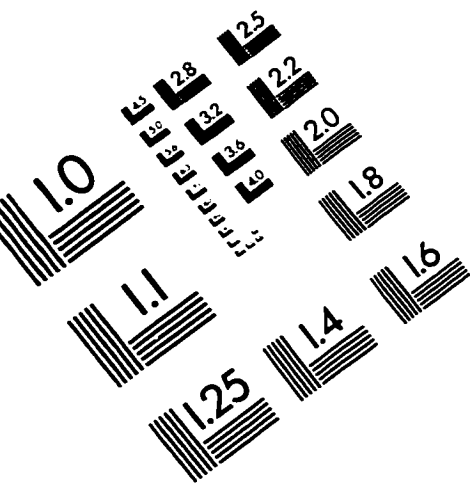
- [80] McCallum, R.A. Learning to use selective attention and short-term memory in sequential tasks. In *Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior: From Animals to Animats* (1996).
- [81] Monahan, G.E. A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science* 28 (1982), 1–16.
- [82] Nilsson, N.J. *Problem Solving Methods in Artificial Intelligence*. McGraw-Hill, New York, 1971.
- [83] Nilsson, N.J. *Principles of Artificial Intelligence*. Tioga Publishing Company, Palo Alto, CA, 1980.
- [84] Papadimitriou, C.H., and Tsitsiklis, J.N. The complexity of Markov decision processes. *Mathematics of Operations Research* 12 (1987), 441–450.
- [85] Parr, R., and Russell, S. Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)* (1995), pp. 1088–1094.
- [86] Parr, R., and Russell, S. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 10: Proceedings of the 1997 Conference* (Denver, CO, 1998), MIT Press.
- [87] Pattipati, K.R., and Alexandridis, M.G. Application of heuristic search and information theory to sequential fault diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics* 20 (1990), 872–887.
- [88] Paz, A. *Introduction to Probabilistic Automata*. Academic Press, New York, 1971.
- [89] Pierskalla, W.P., and Voelker, J.A. A survey of maintenance models: The control and surveillance of deteriorating systems. *Naval Research Logistics Quarterly* 23 (1976), 353–388.
- [90] Platzman, L.K. *Finite Memory Estimation and Control of Finite Probabilistic Systems*. PhD thesis, Massachusetts Institute of Technology, 1977.
- [91] Platzman, L.K. Optimal infinite-horizon undiscounted control of finite probabilistic systems. *Siam Journal of Control and Optimization* 18 (1980), 362–380.
- [92] Platzman, L.K. A feasible computational approach to infinite-horizon partially-observed Markov decision problems. Tech. rep., Georgia Institute of Technology, 1981.
- [93] Puterman, M.L. *Markov Decision Problems*. Wiley, New York, 1994.
- [94] Qi, R. *Decision Graphs: Algorithms and Applications to Influence Diagram Evaluation and High-Level Path Planning Under Uncertainty*. PhD thesis, University of British Columbia, 1994.

- [95] Rosenschein, S.J., and Kaelbling, L.P. A situated view of representation and control. *Artificial Intelligence* 73 (1995).
- [96] Ross, S. Quality control under Markovian deterioration. *Management Science* 17 (1971), 587–596.
- [97] Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Upper Saddle River, NJ, 1995.
- [98] Satia, J.K., and Lave, R.E. Markovian decision processes with probabilistic observation of states. *Management Science* 20 (1973), 1–13.
- [99] Shatkay, H., and Kaelbling, L.P. Learning topological maps with weak local odometric information. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence (AAAI-97)* (1997).
- [100] Singh, S.P., Jaakkola, T., and Jordan, M.I. Model-free reinforcement learning for non-Markovian decision problems. In *Proceedings of the Eleventh International Conference on Machine Learning* (1994), pp. 284–292.
- [101] Smallwood, R.D. The analysis of economic teaching strategies for a simple learning model. *Journal of Mathematical Psychology* 8 (1971), 285–301.
- [102] Smallwood, R.D., and Sondik, E.J. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research* 21 (1973), 1071–1088.
- [103] Sondik, E.J. *The Optimal Control of Partially Observable Markov Processes*. PhD thesis, Stanford University, 1971.
- [104] Sondik, E.J. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research* 26 (1978), 282–304.
- [105] Striebel, C.T. Sufficient statistics in the optimal control of stochastic systems. *Journal of Mathematical Analysis and Applications* 12 (1965), 576–592.
- [106] Sutton, R.S., and Barto, A.G. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.
- [107] Sutton, R.S., Precup, D., and Singh, S. Between MDPs and semi-MDPs: Learning, planning, and representing knowledge at multiple temporal scales. Submitted for publication.
- [108] Tarjan, R. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1 (1972), 146–160.
- [109] Tash, J., and Russell, S. Control strategies for a stochastic planner. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)* (Seattle, WA, 1994), AAAI Press, pp. 1079–1085.

- [110] Teller, A. The evolution of mental models. In *Advances in Genetic Programming*, Jr. K.E. Kinneer, Ed. The MIT Press, 1994, pp. 199–219.
- [111] Thiébaux, S., Hertzberg, J., Shoaff, W., and Schneider, M. A stochastic model of actions and plans for anytime planning under uncertainty. *International Journal of Intelligent Systems* 10 (1995).
- [112] Thomas, L.C., Gaver, D.P., and Jacobs, P.A. Inspection models and their application. *IMA Journal of Mathematics Applied in Business and Industry* 3 (1991), 283–303.
- [113] Washington, R. Incremental Markov-model planning. In *Proceedings of TAI-96: Eighth IEEE International Conference on Tools with Artificial Intelligence* (1996), pp. 41–47.
- [114] Washington, R. Uncertainty and real-time therapy planning: Incremental Markov model approaches. In *Proceedings of the AAAI Spring Symposium on Artificial Intelligence in Medicine* (1996).
- [115] Weld, D.S. Planning-based control of software agents. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)* (1996).
- [116] White III, C.C. Optimal diagnostic questionnaires which allow less than truthful responses. *Information and Control* 32 (1976), 61–74.
- [117] White III, C.C. Procedures for the solution of a finite-horizon partially observed, semi-Markov optimization problem. *Operations Research* 24 (1976), 348–358.
- [118] White III, C.C. Optimal inspection and repair of a production process subject to deterioration. *Journal of the Operational Research Society* 29 (1978), 235–243.
- [119] White III, C.C. Optimal control-limit strategies for a partially observed replacement problem. *International Journal of Systems Science* 10 (1979), 321–331.
- [120] White III, C.C. A survey of solution techniques for the partially observed Markov decision process. *Annals of Operations Research* 32 (1991), 215–230.
- [121] White III, C.C., and Scherer, W.T. Finite-memory suboptimal design for partially observed Markov decision processes. *Operations Research* 42 (1994), 439–455.
- [122] Wiering, M., and Schmidhuber, J. Solving POMDPs with Levin search and EIRA. In *Proceedings of the Thirteenth International Conference on Machine Learning* (1996).
- [123] Witten, I.H. The apparent conflict between estimation and control – a survey of the two-armed bandit problem. *Journal of the Franklin Institute* 301 (1976), 161–189.

- [124] Zhang, N.L., and Lee, S.S. Planning with partially observable Markov decision processes: Advances in exact solution method. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-98)* (Madison, WI, 1998).
- [125] Zhang, N.L., and Liu, W. A model approximation scheme for planning in stochastic domains. *Journal of Artificial Intelligence Research* 7 (1997), 199–230.

IMAGE EVALUATION TEST TARGET (QA-3)



APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved

