University of Kentucky
UKnowledge

University of Kentucky Doctoral Dissertations | Graduate School

2001

Finite Memory Policies for Partially Observable Markov Decision Proesses

Christopher Lusena
University of Kentucky, lusena@cs.uky.edu

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation

Lusena, Christopher, "Finite Memory Policies for Partially Observable Markov Decision Proesses" (2001). *University of Kentucky Doctoral Dissertations*. 323.
https://uknowledge.uky.edu/gradschool_diss/323

This Dissertation is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Doctoral Dissertations by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

ABSTRACT OF DISSERTATION

Christopher Lusena

The Graduate School
University of Kentucky
2001

FINITE MEMORY POLICIES FOR PARTIALLY OBSERVABLE MARKOV DECISION PROCESSES

ABSTRACT OF DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
at the University of Kentucky

By
Christopher Lusena
Lexington, Kentucky
Director: Dr. Judy Goldsmith, Associate Professor of Computer Science
Lexington, Kentucky
2001

ABSTRACT OF DISSERTATION

FINITE MEMORY POLICIES FOR PARTIALLY OBSERVABLE MARKOV DECISION PROCESSES

This dissertation makes contributions to areas of research on planning with POMDPs: complexity theoretic results and heuristic techniques. The most important contributions are probably the complexity of approximating the optimal history-dependent finite-horizon policy for a POMDP, and the idea of heuristic search over the space of FFTs.

Christopher Lusena

July 24, 2001

FINITE MEMORY POLICIES FOR PARTIALLY OBSERVABLE MARKOV DECISION PROCESSES

By

Christopher Lusena

Dr. Judy Goldsmith
Director of Dissertation

Grzegorz Wasilkowski
Director of Graduate Studies

July 24, 2001

RULES FOR THE USE OF DISSERTATIONS

Unpublished dissertations submitted for the Doctor's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the dissertation in whole or in part also requires the consent of the Dean of The Graduate School of the University of Kentucky.

DISSERTATION

Christopher Lusena

The Graduate School
University of Kentucky
2001

FINITE MEMORY POLICIES FOR PARTIALLY OBSERVABLE MARKOV DECISION PROCESSES

DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
at the University of Kentucky

By
Christopher Lusena
Lexington, Kentucky
Director: Dr. Judy Goldsmith, Associate Professor of Computer Science
Lexington, Kentucky
2001

Acknowledgments

I would like to thank:

- My family, parents and grandparents for their emotional and financial support throughout childhood, and graduate school.
- My thesis advisor Judy Goldsmith, for her support, and effort, not only in this thesis or graduate school, but though out my academic career.
- Stefan Bilaniuk, for his advising at Trent, about graduate school, and trivia.
- The faculty of University of Kentucky for their teaching and instruction.
- The staff of the Department of Computer Science for their help in the administration of my graduate career.
- And greatly, Emily Hendren for her help editing this work.

This work supported in part by NSF grant CCR-9315354 and CCR-9610348.[1]

[1]Which is, of course, to thank Uncle Sam for the cash.

Table of Contents

List of Tables

List of Figures

List of Files

lusena01.pdf	.pdf	812 kb

Chapter 1
Introduction

A Markov decision process (MDP) is a model of a controlled stochastic system. For an MDP the controller is assumed to have complete knowledge about the current state of the system. A partially observable MDP (POMDP) is a variant of an MDP where the controller has only partial information from the system. /

MDPs have been used to model phenomena from economics to machine maintenance to ecology (Puterman, 1994), and as such are an extremely useful modeling tool. POMDPs can be used to model many other applications, but the lack of good algorithms has limited their use to a few areas.

In MDPs and POMDPs the decision maker or *agent* performs some *action* that will determine the probabilities of which state the system will be in at the next decision epoch. In general, the most computationally hard problem is finding the action that will maximize the agent's *reward,* which is a one-dimensional measure of the utility of an action. The computational difficulty of the problem depends on the exact phrasing, but the most common version for POMDPs is PSPACE-hard and not ε-approximable.

Most of the current algorithmic effort on POMDPs has gone into solving the PSPACE-hard problem. In the work proposed here, we will study an apparently easier problem that may give better results than the harder one.

1.1 Motivating Applications for Partially Observable Markov Decision Processes

POMDPs have many interesting applications. Unfortunately, POMDPs are not suited for modeling every problem, because of the lack of fast algorithms for various operations. The ideal problem to model with POMDPs is one that requires some or all of the expressiveness of the model, i.e stochastic transitions and partial observations, and one where using a large amount of CPU resources is practical. Several computations are involved in constructing controllers for POMDPs. Computations are required for constructing the controller and for using the controller. Most algorithms are time intensive in the construction phase, but much less resource intensive when applying the controller. Thus POMDPs are useful in many automation and robotic applications. We will present various applications from (Cassandra, 1998b) that meet the criteria discussed above.

The first suggested application area for POMDPs was machine maintenance by Smallwood and Sondik (1973). Here one wishes to decide an operation and maintenance schedule

for a machine on a factory floor. The state is the physical condition of the machine, the actions are operating and maintaining the machine, and the reward function is the value of what is produced less the cost of operation and maintenance. This system is partially observable because to have complete knowledge of the condition of machine, one must take it apart, which is an action of great cost. One can afford to apply the system since factory machines have a very long lifetime, and re-tooling a plant (to change the role of a given machine) may take many weeks and cost millions of dollars. Other similar applications include assembly line control, structural inspection, and elevator control policies. In all these applications the items in question are expensive and are expected to have a long life.

The first area to actually apply POMDPs was robotics, because a person cannot make decisions for an autonomous robot. The most successful applications use multi-level controllers with POMDPs as a middle or top layer. The classical example here is robot navigation: Low-level controllers are currently very good at recognizing the existence of obstacles in a given direction, but not the type of obstacles (e.g., a person or door may appear the same sometimes), and they are good at moving a robot with some accuracy, even if movements have various levels of error associated with them that occur with known probabilities. A mid-level POMDP controller can perform simple assigned tasks with a high degree of certainty, e.g. moving from point A to point B, which it does by assigning actions and getting observations from the low level controller. The purpose of these multi-level controllers is to simplify the problem into pieces that are solvable by an application. Although the complete problem could be modeled as a POMDP, the resulting model would be too large to solve by today's methods.

Many more application areas can be modeled by POMDPs, including various economic problems, machine vision problems, and biological problems. For a recent survey, see (Cassandra, 1998b).

1.2 Solving Partially Observable Markov Decision Processes

Three major problems are involved in using MDPs or POMDPs to solve an application. The first is constructing the MDP or POMDP, the second is finding a desirable policy for the controller, and the third is to evaluate a given policy for a given model. We are interested in the second and third problems. We are assuming that a model of the real world problem will be delivered in a correct form. Solving the first problem is usually done by either constructing the model directly via observations and experiments (Puterman, 1994) or by reinforcement learning techniques (Sutton & Barto, 1998).

In the case where the agent has perfect information about the state of the system, there exist algorithms to evaluate a policy or to find the best policy in time polynomial in the size of the state space (Puterman, 1994). Unfortunately, the number of states of a system is often exponential with respect to the number of the features of the system; for instance, if each of n machines on an assembly line has three states, then an MDP modeling the complete assembly would have 3^n states. Since large systems, ones with too many states to handle effectively with polynomial time algorithms, often have a great deal of apparent structure, there is research interest in factored and compressed representations (Boutilier, Dean, & Hanks, 1995; Boutilier, Dearden, & Goldszmidt, 1995; Boutilier & Poole, 1996; Boutilier, Dean, & Hanks, 1999; Mundhenk, Goldsmith, Lusena, & Allender, 2000). The techniques we study rely on polynomial time solution and manipulation of MDP problems. If the necessary sub-algorithms are reasonably fast, then the proposed work will extend to factored representations. Thus we will not discuss how to model truly large systems, since this is still problematic even in the MDP case.

The techniques to find the optimal policy of an MDP vary depending on the evaluation criterion of the policy. In the case where the system is to run for some fixed finite amount of time, two common criteria exist: *total expected reward* and *total discounted expected reward.* In the latter case, it is assumed that costs and profits of each action decrease in relative value by some discount factor γ as time continues. In either case an optimal controller can be found by a dynamic programming technique that Puterman (1994, page 92) calls backward induction (see Section 4.2.1.2). Evaluating a policy is done via another dynamic programming technique that is best called forward induction (Puterman, 1994, page 80) (see Section 4.2.1.1). Forward induction is only one order less in complexity than backward induction, so forward induction's usefulness is limited to the case where one does not wish to know and/or use the optimal policy, since finding the optimal policy is almost as fast as evaluating a policy.

When one wishes to model a system in perpetuity, the choice of policy evaluation metric becomes cloudier. The total expected reward may be infinite. Thus one of several other metrics comes into play. For the standard metric of total discounted reward, there exist linear programs that are polynomial in the size of the MDP whose solutions represent the optimal policy and its value (Puterman, 1994). (See Chapters 2 and 4).

Unfortunately, POMDPs are harder to solve than MDPs for all three major problems: modeling, optimal policy construction and policy evaluation. A difficulty for the computational problems lies in what information the agent must keep around in order to have enough information to act optimally; this information is called a *sufficient statistic.* For MDPs just

the current state is a sufficient statistic, but for POMDPs one may need much more information, for instance the complete history of observations seen and actions performed by the agent. The most popular sufficient statistic for POMDPs is the *belief state*, where the agent stores the probability that the system is in each state, given the sequence of observations it has made and actions it has taken. Hence the belief state is a vector of probabilities of dimension the size of the state space. The set of all possible belief states forms a space called the *belief space.* The belief state was shown to be a sufficient statistic by Aoki (1965).

Suppose that one knows the *value of the optimal policy* for a given POMDP at every point in the belief space. Then one can compute what the optimal policy will do next by a simple maximizing equation over all actions (see Chapter 4 for more details). Astrom (1965) showed that the value function over the belief space is convex, and Sondik (1971) showed that over a finite horizon it is piecewise linear and representable by a finite number of vectors. These two facts form the basis for a class of algorithms called *value iteration.* These algorithms first form either the value function or an approximation to it, then use the computed value to induce the policy. In the infinite case under discounted rewards, Sondik showed that one can use a finite horizon value function as a reasonable approximation of the infinite one.

The value iteration algorithms are further divided into *exact* and *approximation* algorithms, depending on whether they construct optimal value functions or approximations to optimal value functions. These algorithms use a set of vectors to represent the value of the optimal policy. The first exact algorithm was Sondik's one-pass algorithm (Sondik, 1971) in his thesis, with some proposed improvements in (Smallwood & Sondik, 1973), which proved to be in error (Lovejoy, 1991b). Cheng's linear support (Cheng, 1988) used a new set of conditions to gain an improvement in speed over Sondik's. In both cases the representation is built up from the empty set to a set with zero error. In contrast, Monahan's exhaustive algorithm (Monahan, 1982) starts with a large correct set and removes vectors as the process goes along. Lark (Lark III, 1990; White, 1991) uses a different pruning method over Monahan to gain a speedup, though his method as stated has a minor error that was corrected by Littman (Littman, 1996).

This brings us to the most modern exact algorithms, the witness algorithms (Littman, 1994b; Cassandra, Kaelbling, & Littman, 1994, 1995; Cassandra, 1994; Littman, 1996; Cassandra, 1998a) and incremental pruning (IP) (Zhang & Liu, 1997a; Cassandra, Littman, & Zhang, 1997; Zhang & Lee, 1998). These algorithms both build and prune the vector set at the same time. These algorithms proceed iteratively over time, finding the new vector set from the old one. For each possible vector in a new time step, a linear program must be

solved. The number of variables of the linear program is the size of the original POMDP's state space, and the number of constraints is the number of vectors in the old vector set. The number of vectors tends to grow exponentially with respect to the number of iterations of this process. The major improvement of IP over the witness algorithm is in decreasing the number of constraints in the linear programs via a reordering of building and pruning phases. Recent work has been done on further decreasing the number of constraints without a great increase in additional overhead (Zhang & Lee, 1998).

Unfortunately, even the champion exact algorithm, IP, cannot solve many interesting problems without excessive resources (Lovejoy, 1991b; Cassandra, 1998a). Most interesting problems have more than 50 states. Most people are unwilling to wait the many weeks of machine time that the smallest and simplest of such problems would need. All these algorithms are memory intensive as well. Thus the necessary hardware is too costly for most medium-sized and larger problems, regardless of time constraints. Therefore we need heuristic and approximation approaches.

Approximation algorithms can be divided into at least two categories: those that attempt to approximate the value function (value iteration) and those that attempt to find a policy (policy iteration). Value iteration has several methods, a few of which are currently considered the best methods for producing controllers. In grid methods, the value function is computed for a small number of points in the belief space, and then interpolated elsewhere. The grid points can be chosen to be evenly spaced (Lovejoy, 1991a), induced by some choice of finite history sequences (Platzman, 1977; White & Scherer, 1994) or by a heuristic adaptation (Hauskrecht, 1997). A second method is using the value function of a controller, such as a finite state controller (Hansen, 1998a; Sondik, 1971). For a comparison of how various value iteration techniques work, see Hauskrecht (1997).

Policy iteration, where one directly constructs a controller instead of the value function, is a lesser-studied form than value iteration, because it is hard or impossible for these methods to compute any guarantee on the performance of the controller relative to the optimal controller. The new algorithms proposed can be thought of as being policy iterative, since we are constructing controllers. Sondik (1971) first thought of a finite memory controller as a mapping of regions of the belief space to actions. Unfortunately, according to Hansen (1998a), the method proposed is complicated, involving three data structure conversions and computations per iteration. Hansen (1998a) has also studied finite memory controllers. He chooses finite state controllers as the representation, which have the advantages of being easy to manipulate and representing piecewise linear and convex value functions over the belief space. Thus his work could be also used for value iteration.

The last major group of controller constructions are the heuristic controllers, which are simple rules to decide on the action depending on the belief state. These are currently the most scalable methods, since they use only polynomial time algorithms. An example of such heuristic controller is Most Likely State: First one solves the MDP one gets by assuming the system is fully observable, and then the controller just chooses the action that corresponds to the state with the greatest probability in the belief state. Cassandra (1998a, Chapter 6) discusses heuristic controllers in great detail. He found that over a relatively small data set all those that he tried performed well some of the time, but none work well all the time.

1.3 Overview of Dissertation

The major contributions of this thesis are the complexity results about finding optimal and approximately optimal POMDP policies (Chapter 3), and the experimental results for heuristic construction of POMDP policies (Chapter 6 and 7).

We approach this work by first giving definitions and theoretical results and then proceeding to algorithmic solutions. In order to discuss algorithmic solutions we first need some definitions and theoretical results. The majority of prior work has been algorithmic in nature. We begin with definitions of MDPs, POMDPs, policies, and complexity classes in Chapter 2, then give complexity results in Chapter 3. We then give a survey of older work on MDPs and POMDPs, both to put our work into perspective and to introduce ideas and techniques used in our work. Chapter 5 discusses free finite table policies; this and the chapters describing the experiments are, of course, original work.

Chapter 2
Definitions

While several definitions of Markov decision processes (MDPs) and partially observable MDPs (POMDPs) appear in the literature, throughout we will use the definitions of (Mundhenk et al., 2000), with some modification for more generality.

2.1 Partially Observable Markov Decision Processes

A POMDP describes a world by its states and the consequences of actions on the world. It is denotes as a tuple $M = (\mathcal{S}, s_0, \mathcal{A}, \mathcal{O}, t, o, r)$, where

- $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{O}$ are finite sets of *states*, *actions*, and *observations*;
- $s_0 \in \mathcal{S}$ is the *initial state*; occasionally one may want s_0 to be a function of the starting probabilities for each state, $s_0 : \mathcal{S} \to [0, 1]$ the likelihood of starting in a given state. Therefore $\sum_{s \in \mathcal{S}} s_0(s) = 1$. Typical values for s_o as a function are, 0, 1, and $1/|\mathcal{S}|$;
- $t : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the *state transition function*, where $t(s, a, s')$ is the probability that state s' is reached from state s on action a (note that $\Sigma_{s' \in \mathcal{S}} t(s, a, s') \in \{0, 1\}$ for every $s \in \mathcal{S}, a \in \mathcal{A}$);
- $o : \mathcal{S} \to \mathcal{O}$ is the *observation function* , where $o(s)$ is the observation made in state s; and
- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{Q}$ is the *reward function*, where $r(s, a)$ is the reward gained by taking action a in state s; $\mathbb{Q}$ is the set of rational numbers.[1]

If states and observations are identical, i.e. $\mathcal{O} = \mathcal{S}$ and o is the identity function, then the POMDP is called *fully-observable* or just an MDP. Otherwise it is called *(deterministically) partially observable*. POMDPs having a probabilistic observation function can be transformed into POMDPs having a deterministic observation function via expansion of the state space.

Throughout the paper we mainly consider finite-horizon MDPs. We will denote the horizon as h, with $h \in \mathbb{N} - \{0\}$. Many results and much of the proposed work are extendible to infinite horizon policies. We will discuss the basics of this separately. The inclusion of s_o in the definition of a POMDP is somewhat controversial, since this defines an initial

[1] In theory one may use $\mathbb{R}$ (the real numbers), but we wish to restrict ourselves to representable numbers.

belief state. However, we believe that this is a modeling issue: Once one starts to control a system, one will have a belief state; thus our work will apply, although in some instances FFT algorithms' output may not be as reusable as others.

2.2 Policies

A policy specifies the controller's choice of actions depending on its observations and its internal state, which may depend on the history of the computation or the time step.

A *stationary policy* π_s (for M) is a function $\pi_s : \mathcal{O} \to \mathcal{A}$ mapping each observation ω to an action $\pi(\omega)$.

A *time-dependent policy* π_t is a function $\pi_t : \mathcal{O} \times \mathcal{H} \to \mathcal{A}$ mapping each observation-time pair to an action, where $\mathcal{H} = \{i \in \mathbb{N} : i < h\}$.

A *history-dependent policy* π_h is a function $\pi_h : \mathcal{O}^* \to \mathcal{A}$, mapping strings of observations to actions.

A *free finite table (FFT) policy* over a set $\mathcal{M}$ is a function $\pi : \mathcal{O} \times \mathcal{M} \to \mathcal{A} \times \mathcal{M}$. The controller is given fixed finite read/write memory, which it may reference and update with every action it takes. Note that if the controller has n bits of memory, then it has 2^n states of memory. A finite memory policy over MDPM with memory set $\mathcal{M} = \{0, \ldots m-1\}$ is a stationary policy for a new MDPM' with:

$$\begin{aligned}
\mathcal{S}' &= \mathcal{S} \times \mathcal{M} \\
s'_0 &= (s_0, 0) \\
A' &= \mathcal{A} \times \mathcal{M} \\
\mathcal{O}' &= \mathcal{O} \times \mathcal{M} \\
t'((u, m_1), (a, m_2), (v, m_3)) &= \begin{cases} t(u, a, v) & \text{if } m_2 = m_3 \\ 0 & \text{otherwise} \end{cases} \\
o'((s, m)) &= (o(s), m) \quad \text{and} \\
r'((s, m_1), (a, m_2)) &= r(s, a).
\end{aligned}$$

A restriction may be put on the controller's use of its memory states. This restriction is given by the function $\Re : \mathcal{O} \times \mathcal{M} \to \mathbb{P}(\mathcal{M})$. For the given observation-memory pair the controller may only change its memory to an element of some subset of the set of possible memories $\mathcal{M}$. There are several examples of this: for instance, remembering a finite sequence of observations (*finite-history policy*). To model a time-dependent policy we can set $\Re((o, i)) = \{i + 1\}$, and $|\mathcal{M}| = h$. If $\Re((o, m))$ is always $\mathcal{M}$, then we call such a policy a free finite memory policy, and a restricted finite-memory policy otherwise. In the restricted

case r' is redefined as:

$$r'((s,m_1),(a,m_2)) = \begin{cases} r(s,a) & \text{if } m_2 \in \Re(o'(s,m_1)) \\ -\infty & \text{otherwise,} \end{cases}$$

guaranteeing that any policy with a reasonable expected reward avoids illegal assignments.[2] Using appropriate choices of memory size and restrictions, one can simulate many different types of policies.

2.3 Policy Evaluation

Let $M = (\mathcal{S}, s_0, \mathcal{A}, \mathcal{O}, t, o, r)$ be a POMDP, with $s_0(\sigma)$ being the probability of starting in state σ.

A *trajectory θ of length m for M* is a sequence of states $\theta = \sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_m$ ($m \geq 0$, $\sigma_i \in \mathcal{S}$).We use $T_k(s)$ to denote the set of length-k trajectories which end in state s.

The expected reward obtained in state s after exactly k steps under policy π is the reward obtained in s by taking the action specified by π, weighted by the probability that s is actually reached after k steps,

- $r(s,k,\pi) = r(s,\pi(o(s))) \cdot \sum_{(\sigma_0,\ldots,\sigma_k)\in T_k(s)} s_0(\sigma_0) \prod_{i=1}^{k} t(\sigma_{i-1}, \pi(o(\sigma_{i-1})), \sigma_i)$, if π is a *stationary policy;*
- $r(s,k,\pi) = r(s,\pi(o(s),k)) \cdot \sum_{(\sigma_0,\ldots,\sigma_k)\in T_k(s)} s_0(\sigma_0) \prod_{i=1}^{k} t(\sigma_{i-1}, \pi(o(\sigma_{i-1}), i-1), \sigma_i)$, if π is a *time-dependent policy;* and
- $r(s,k,\pi) = \sum_{(\sigma_0,\ldots,\sigma_k)\in T_k(s)} r(s,\pi(o(\sigma_0)\cdots o(\sigma_k))) \cdot s_0(\sigma_0) \prod_{i=1}^{k} t(\sigma_{i-1}, \pi(o(\sigma_0)\cdots o(\sigma_{i-1})), \sigma_i)$, if π is a *history-dependent policy.*

A POMDP may behave differently under optimal policies for each type of policy. The quality of a policy is determined by its *performance*, i.e. by the expected rewards accrued. We distinguish between different performance metrics for POMDPs that run for a finite number of steps and those that run indefinitely.

- The *finite-horizon performance of a policy* π for POMDPM is the expected sum of rewards received during the first h steps by following the policy π, i.e., $\mathrm{perf}_f(M,\pi) = \sum_{i=0}^{h-1} \sum_{s\in\mathcal{S}} r(s,i,\pi)$. (Other work assumes that the horizon is *poly*$(|M|)$, instead of $|M|$. This does not change the complexity of any of our problems.)

[2] Of course $-\infty$ is not representable, so in practice one would use a sufficiently large negative value.

- The infinite-horizon *total discounted performance* gives rewards obtained earlier in the process a higher weight than those obtained later. For $0 < \gamma < 1$, the total γ-discounted reward is defined as $\text{perf}^{\gamma}_{td}(M, \pi) = \sum_{i=0}^{\infty} \sum_{s \in \mathcal{S}} \beta^i \cdot r(s, i, \pi)$.

- The infinite-horizon *average performance* is the limit of all rewards obtained within n steps divided by n, for n going to infinity:[3] $\text{perf}_{av}(M, \pi) = \lim_{n \to \infty} \frac{1}{n} \text{perf}_f(M, n, \pi)$.

Let perf be any of these performance metrics, and let α be any policy type, either stationary, time-dependent, or history-dependent. The *α-value* $\text{val}_\alpha(M)$ of M (under the metric chosen) is the maximal performance of any policy π of type α for M, i.e. $\text{val}_\alpha(M) = \max_{\pi \in \Pi_\alpha} \text{perf}(M, \pi)$, where Π_α is the set of all α policies.

2.4 Partially Observable Markov Decision Processes Value Problem

For formal mathematical methods we need a definition of language problem for POMDPs. We use the POMDP Value Problem, which is: "Given a POMDP M, a starting state s_0, a horizon k, and a value V does there exists a policy π, such that $\text{perf}(M, s_0, k, \pi) \geq V$?" There are many parameters, that change the nature of this problem:

- The observation of the POMDP: Full, Partial, None;

- The range of the reward function: ≥ 0, (≤ 0), any;

- The type of policy one is looking for: Stationary, Time, History, Finite Memory;

- The evaluation criteria for policies: Total expected reward, total discounted expected reward, average expected reward.

Some times we will also be interested in the value of the optimal policy, which is referred to as the value of the given POMDP. The language problem is also sometimes called the POMDP Policy Existence Problem.

2.5 Complexity Classes

For definitions of complexity classes, reductions, and standard results from complexity theory, we refer to (Papadimitriou, 1994). In the interest of completeness, in this section we give a short description of the complexity classes we use in this work.

[3] If this limit is not defined, the performance is defined as a lim inf.

It is important to note that most of the classes we consider are *decision classes.* This means that the problems in these classes are "yes/no" questions. Thus, for instance, the traveling salesperson problem is in NP in the form, "Is there a TSP tour within budget b?" The question of finding the *best* TSP tour for a graph is technically not in NP, although the decision problem can be used to find the optimal value via binary search. Although there is an optimization class associated with NP, there are not common optimization classes associated with all of the decision classes we reference. Therefore, we have phrased our problems as decision problems in order to use known complete problems for these classes.

The sets decidable in polynomial time form the class P. Decision problems in P are often said to be "tractable," or more commonly, problems that cannot be solved in polynomial time are said to be intractable. A standard complete problem for this class is the *circuit value problem (CVP):* Given a Boolean circuit and an input, does the circuit output a 1?

The most common PSPACE-complete problem is QBF, the set of true quantified Boolean formulas with no free variables. Another variant that is also complete for PSPACE is stochastic satisfiability, where QBF-style formulas alternate existential and random quantifiers.

For the complexity classes mentioned, the following inclusions hold:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE}.$$

2.6 Approximability

We show in Chapter 3 that computing optimal policies for POMDPs is hard. Instead of trying to compute an optimal policy, we might wish to compute a policy guaranteed to have a value that is at least a large fraction of the optimal value.

A polynomial-time algorithm computing such a nearly optimal policy is called an ε-approximation (for $0 \leq \varepsilon < 1$), where ε indicates the quality of the approximation in the following way. Let A be a polynomial-time algorithm which for every POMDPM computes an α-policy $A(M)$. Notice that $\mathrm{perf}(M, A(M)) \leq \mathrm{val}_\alpha(M)$ for every M. The algorithm A is called an ε-*approximation* if for every POMDPM,

$$\left| \frac{\mathrm{val}_\alpha(M) - \mathrm{perf}(M, A(M))}{\mathrm{val}_\alpha(M)} \right| \leq \varepsilon \ .$$

(See, e.g., Papadimitriou, 1994 for more detailed definitions.) Approximability distinguishes NP-complete problems: Problems exist which are ε-approximable for all ε, for certain ε, or for no ε (unless P = NP). Note that this definition of ε-approximation requires that $\mathrm{val}_\alpha(M) \geq 0$. If a policy with positive performance exists, than every approximation algorithm yields

such a policy, because a policy with performance 0 or smaller cannot approximate a policy with positive performance. Hence, any approximation straightforwardly solves the decision problem.

An approximation scheme yields an ε-approximation for arbitrary $\varepsilon > 0$. If there is a polynomial-time algorithm that on input POMDP M and ε outputs an ε-approximation of the value in time polynomial in the size of M, then we say the problem has a *Polynomial-Time Approximation Scheme (PTAS)*. If the algorithm runs in time polynomial in the size of M and $\frac{1}{\varepsilon}$, the scheme is a *Fully Polynomial-Time Approximation Scheme (FPTAS)*. All of the PTASs constructed here are FPTASs; we state the theorems in terms of PTASs because that gives stronger results in some cases, and because we do not explicitly analyze the complexity in terms of $\frac{1}{\varepsilon}$.

If there is a polynomial-time algorithm that outputs an approximation, v, to the value μ of M ($\mu = \mathrm{val}_\alpha(M)$) with $\mu \geq v \geq \mu - k$, then we say that the problem has a *k-additive approximation algorithm.*

Chapter 3
Theoretical Results

In this chapter, we consider the computational complexity of computational problems associated with computing policies for Markov decision processes (MDPs) and partially observable MDPs (POMDPs). The first work on this should be attributed to Bellman (1957), although his analysis did not explicitly consider computational complexity. The first to explicitly consider it were Papadimitriou and Tsitsiklis (1986).

Many of the theorems in this chapter appeared in (Mundhenk et al., 2000; Lusena, Goldsmith, & Mundhenk, 2001). We also reference (Papadimitriou & Tsitsiklis, 1987; Madani, Hanks, & Condon, 1999).

This chapter is divided into results on MDPs and POMDPs, and the POMDP section is further subdivided into results about stationary, time-dependent, and history-dependent policies. Within each subsection, we consider policy existence problems and approximability for both the finite and infinite horizon cases.

3.1 Markov Decision Processes

Bellman (1957) formulated MDPs and provided outlines of algorithms that run in polynomial time in the size of the state space. So parts of complexity theoretic results are really due to him, even though such terminology postdates his work by a decade. He both gave results and showed that various other results are not interesting (see Section 4.2). The first explicit complexity results were by Papadimitriou and Tsitsiklis (1986).

In particular, Bellman (1957) showed that one can find and evaluate stationary policies for MDPs under total and discounted metrics for the finite horizon, and under the discounted metric in the infinite horizon. Some complexity results for these problems are implicit in his work, though showing that the infinite horizon, discounted policy existence problem is in P had to wait until linear programming was shown to be in P.

Theorem 3.1.1. *The time-dependent and history-dependent policy existence problems for fully-observable MDPs in the finite horizon are* P*-complete.*

Because the proof is a straightforward modification of a proof in (Papadimitriou & Tsitsiklis, 1987), we omit it here.

In the infinite-horizon case, one need only consider stationary policies, since they are optimal (the optimal time-dependent and history-dependent policies are not better).

Theorem 3.1.2. *The stationary policy existence problem for fully-observable MDPs under discounted rewards in the infinite horizon is P-complete.*

Proof. Papadimitriou and Tsitsiklis (1987) show that it is P-hard to decide whether a discounted reward time-dependent policy with value less than zero exists for an MDP in the finite horizon. It is a simple matter to change their negative rewards to positive and make some other small adjustments to their reduction from the circuit value problem, to show that our problem is also P-complete.

Bellman (1957) shows that finding an optimal stationary policy for an MDP is reducible to a linear programming problem of similar size. Thus, the problem is shown to be in P.

□

Corollary 3.1.3. *The stationary policy existence problem for fully-observable MDPs under average rewards in the infinite horizon is P-hard.*

3.2 Partially Observable Markov Decision Processes

POMDPs were first formalized in (Sondik, 1971). In this work Sondik gave a formal definition and an algorithm to find history dependent policies, though without computing bounds on the running time of said algorithm. Littman (1996) showed that each iteration of the algorithm runs in exponential time.

The first complexity results were again by Papadimitriou and Tsitsiklis (1987). They proved that in the finite horizon finding the optimal time-dependent policy is NP-complete and that finding a history-dependent policy is PSPACE-complete. Further work has been done in (Mundhenk et al., 2000; Madani et al., 1999; Lusena et al., 2001)

3.2.1 Stationary Policies

Theorem 3.2.1. *The stationary policy existence problem for POMDPs with nonnegative rewards over the finite horizon is* NP*-complete.*

Proof. Membership in NP is straightforward, because a policy can be guessed and evaluated in polynomial time. To show NP-hardness, we reduce the NP-complete satisfiability problem ϕ to it. Let $\phi(x_1, \ldots, x_n)$ be such a formula with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$, where clause $C_j = (l_{v(1,j)} \vee l_{v(2,j)} \vee l_{v(3,j)})$ for $l_i \in \{x_i, \neg x_i\}$. We say that variable x_i *appears in* C_j *with signum* 0 *(resp.* 1*)* if $\neg x_i$ (resp. x_i) is a literal in C_j. Without loss of generality, we assume that every variable appears at most once in each clause. The idea is to construct a POMDP $M(\phi)$ having one state for each appearance of a variable in a clause. The set

of observations is the set of variables. Each action corresponds to an assignment of a value to a variable. The transition function is deterministic. The process starts with the first variable in the first clause. If the action chosen in a certain state satisfies the corresponding literal, the process proceeds to the first variable of the next clause, or with reward 1 to the final state, if all clauses were considered. If the action does not satisfy the literal, the process proceeds to the next variable of the clause, or with reward 0 to the final state. The partition of the state space into observation classes guarantees that the same assignment is made for every appearance of the same variable. Therefore, the value of $M(\phi)$ equals 1 iff ϕ is satisfiable.

Formally, from ϕ, we construct a POMDP $M(\phi) = (\mathcal{S}, s_0, \mathcal{A}, \mathcal{O}, t, o, r)$ with

$$\mathcal{S} = \{(i,j) \mid 1 \le i \le n, 1 \le j \le m\} \cup \{F, T\}$$

$$s_0 = (v(1,1),1), \qquad \mathcal{A} = \{0,1\}, \qquad \mathcal{O} = \{x_1, \ldots, x_n, F, T\}$$

$$t(s,a,s') = \begin{cases} 1, & \text{if } s = (v(i,j),j), s' = (1,j+1), j < m, 1 \le i \le 3, \\ & \quad \text{and } x_{v(i,j)} \text{ appears in } C_j \text{ with signum } a \\ 1, & \text{if } s = (v(i,m),m), s' = T, 1 \le i \le 3, \\ & \quad \text{and } x_{v(i,m)} \text{ appears in } C_m \text{ with signum } a \\ 1, & \text{if } s = (v(i,j),j), s' = (v(i+1,j),j), 1 \le i < 3, \\ & \quad \text{and } x_{v(i,j)} \text{ appears in } C_j \text{ with signum } 1-a \\ 1, & \text{if } s = (v(3,j),j), s' = F, \\ & \quad \text{and } x_{v(3,j)} \text{ appears in } C_j \text{ with signum } 1-a \\ 1, & \text{if } s = s' = F \text{ or } s = s' = T \\ 0, & \text{otherwise} \end{cases}$$

$$r(s,a) = \begin{cases} 1, & \text{if } t(s,a,T) = 1, s \ne T \\ 0, & \text{otherwise} \end{cases}, \qquad o(s) = \begin{cases} x_i, & \text{if } s = (i,j) \\ T, & \text{if } s = T \\ F, & \text{if } s = F \end{cases}.$$

Note that all transitions in $M(\phi)$ are deterministic, and every trajectory has value 0 or 1. There is a correspondence between policies for $M(\phi)$ and assignments of values to the variables of ϕ, such that policies under which $M(\phi)$ has value 1 correspond to satisfying assignments for ϕ, and vice versa. □

Several corollaries immediately follow from this proof.

Corollary 3.2.2. *The stationary policy existence problem for POMDPs over the finite horizon is* NP*-complete.*

Note that problem considered in Theorem 3.2.1 is a special case of the problem considered in this corollary. Therefore, NP-hardness follows immediately. Because we are considering

deterministic observations, every stationary policy for a POMDP is also a stationary policy for the MDP one gets by replacing the observation function with the identity function. Therefore, the problem is in NP because policy evaluation for stationary policies for MDPs is in P. (See Section 4.2.)

Corollary 3.2.3. *The stationary policy existence problem for POMDPs over the infinite horizon with discounted rewards is NP-complete.*

Corollary 3.2.4. *The stationary policy existence problem for POMDPs over the infinite horizon with average rewards is NP-hard.*

This follows by modifying the proof of Theorem 3.2.1. Let the transition from T to T get reward 1. From this, we see that in both the infinite-horizon models (discounted and average reward), the value of the optimal policy is greater than zero if and only if there is a satisfying assignment for original formula ϕ.

Given that finding optimal policies is, in various cases, probably infeasible, we next ask whether there are good approximation algorithms for policies in these cases. The answer comes back, "No." (Unless, of course, some surprising complexity collapse occurs.)

Theorem 3.2.5. *Let $0 \leq \varepsilon < 1$. An optimal stationary policy for POMDPs with non-negative rewards in the finite horizon is ε-approximable if and only if* P = NP.

Proof. The stationary value of a POMDP can be calculated in polynomial time by a binary search using an oracle for the stationary policy existence problem for POMDPs. The number of bits to be calculated is polynomial in the size of M. Knowing the value, we can try to fix an action for an observation. If the modified POMDP still achieves the value calculated before, we can continue with the next observation, until a stationary policy is found which has the optimal performance. This algorithm runs in polynomial time with an oracle solving the stationary policy existence problem for POMDPs. Since the oracle is in NP by Theorem 3.2.1, the algorithm runs in polynomial time if P = NP.

Now, assume that A is a polynomial-time algorithm that ε-approximates the optimal stationary policy for some ε with $0 \leq \varepsilon < 1$. We show that this implies that P = NP by showing how to solve the NP-complete problem 3SAT. As in the proof of Theorem 3.2.1, given an instance ϕ of 3SAT, we construct a POMDP $M(\phi)$. The only change to the reward function of the POMDP constructed in the proof of Theorem 3.2.1 is to make it a POMDP with positive performances. Now reward 1 is obtained if state F is reached, and reward $\lceil \frac{2}{1-\varepsilon} \rceil$ is obtained if state T is reached. Hence ϕ is satisfiable if and only if $M(\phi)$ has value $\lceil \frac{2}{1-\varepsilon} \rceil$.

Assume that policy π is the output of the ε-approximation algorithm A. If ϕ is satisfiable, then $\text{perf}(M(\phi), \pi) \geq (1-\varepsilon) \cdot \frac{2}{1-\varepsilon} = 2 > 1$. Because the performance of every policy for $M(\phi)$ is either 1 if ϕ is not satisfiable, or $\lceil \frac{2}{1-\varepsilon} \rceil$ if ϕ is satisfiable, it follows that π has performance > 1 if and only if ϕ is satisfiable. So, in order to decide $\phi \in$ 3SAT, we can construct $M(\phi)$, run the approximation algorithm A on it, take its output π and calculate $\text{perf}(M(\phi), \pi)$. That output shows whether ϕ is in 3SAT. All these steps are polynomial time bounded computations. It follows that 3SAT is in P, and hence P = NP. □

Of course, the same nonapproximability result holds for POMDPs with positive and negative rewards.

Corollary 3.2.6. *Let $0 \leq \varepsilon < 1$. Any optimal stationary policy for POMDPs under the total or discounted finite-horizon metrics is ε-approximable if and only if* P = NP.

Corollary 3.2.7. *Let $0 \leq \varepsilon \leq 1$. Any optimal stationary policy for a POMDP under the discounted infinite-horizon metric is ε-approximable if and only if* P = NP.

The modifications of the construction from Theorem 3.2.5 are similar to the modifications used in the previous corollaries.

Next, we consider time-dependent policies. The time-dependent policy existence problem for unobservable MDPs turns out to be NP-complete.

3.2.2 Time-Dependent Policies

Theorem 3.2.8. *The time-dependent finite-horizon policy existence problem for unobservable MDPs is* NP*-complete.*

Papadimitriou and Tsitsiklis proved a similar theorem (Papadimitriou & Tsitsiklis, 1987). Their MDPs had only non-positive rewards, and their formulation of the decision problem was whether there is a policy with reward 0. However, our result can be proven by a proof very similar to theirs showing a reduction from 3SAT.

Proof. That it is in NP follows from the fact that a policy with performance > 0 can be guessed and checked in polynomial time. NP-hardness follows from the following reduction from 3SAT. At the first step, a clause is chosen randomly. At step $i+1$, the assignment of variable i is determined. Because the process is unobservable, it is guaranteed that each variable gets the same assignment in all clauses. If a clause is satisfied by this assignment, it will gain reward 1; if not, the reward will be $-m$, where m is the number of clauses of

the formula. Therefore, if all clauses are satisfied, the time-dependent value of the MDP is positive; otherwise, the value is negative.

We formally define the reduction. Let ϕ be a formula with n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$. We say that x_i appears in C_j with signum 1 if $x_i \in C_j$, and with signum 0 if $\neg x_i \in C_j$. Define the unobservable MDP $M(\phi) = (\mathcal{S}, s_0, \mathcal{A}, t, r)$ where

$$\begin{aligned}
\mathcal{S} &= \{(i,j) \mid 1 \le i \le n, 1 \le j \le m\} \cup \{s_0, T, F\} \\
\mathcal{A} &= \{0,1\} \\
t(s,a,s') &= \begin{cases}
\frac{1}{m}, & \text{if } s = s_0, a = 0, s' = (1,j), 1 \le j \le m \\
1, & \text{if } s = (i,j), s' = T, \ x_i \text{ appears in } C_j \text{ with signum } a \\
1, & \text{if } s = (i,j), s' = (i+1,j), i < n, \\
 & \qquad x_i \text{ doesn't appear in } C_j \text{ with signum } a \\
1, & \text{if } s = (n,j), s' = F, \ x_n \text{ doesn't appear in } C_j \text{ with signum } a \\
1, & \text{if } s = s' = F \text{ or } s = s' = T, a = 0 \text{ or } a = 1 \\
0, & \text{otherwise}
\end{cases} \\
r(s,a) &= \begin{cases}
1, & \text{if } t(s,a,T) > 0 \text{ and } s \neq T \\
-m, & \text{if } t(s,a,F) > 0 \text{ and } s \neq F \\
0, & \text{otherwise.}
\end{cases}
\end{aligned}$$

The correctness of the reduction follows by the above discussion. □

For the infinite horizon, this problem is *much* harder. The following theorem is due to Madani et al. (1999).

Theorem 3.2.9. *The policy existence problem for time-dependent policies for POMDPs under the infinite-horizon discounted or average reward criterion is undecidable.*

Unsurprisingly, given the pattern of previous results, we show that this problem is not likely to be approximable, even in the finite-horizon case.

Theorem 3.2.10. *Let $0 \le \varepsilon < 1$. Any optimal time-dependent, finite-horizon policy for unobservable MDPs with non-negative rewards is ε-approximable if and only if* P = NP.

Proof. We give a reduction from 3SAT with the following properties. For a formula ϕ with m clauses, we show how to construct an unobservable MDP$M_\varepsilon(\phi)$ with value 1 if ϕ is satisfiable, and with value $< (1-\varepsilon)$ if ϕ is not satisfiable. Therefore, an ε-approximation could be used to distinguish between satisfiable and unsatisfiable formulas in polynomial time.

For formula ϕ, we first show how to construct an unobservable $M(\phi)$ from which $M_\varepsilon(\phi)$ will be constructed. $M(\phi)$ simulates the following strategy. At the first step, one of the m clauses is chosen uniformly at random with probability $\frac{1}{m}$. At step $i+1$, the assignment

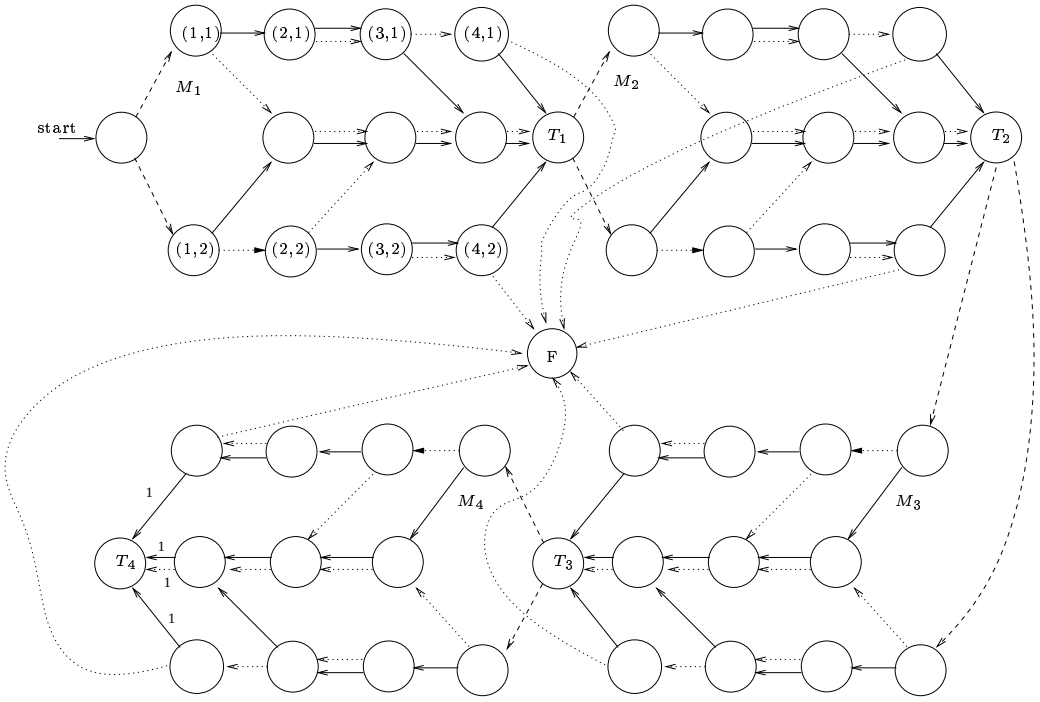


Figure 3.1: An example unobservable MDP for $\phi = (\neg x_1 \vee x_3 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee x_4)$

of variable i is determined. Because the process is unobservable, it is guaranteed that each variable gets the same assignment in all clauses, because its value is determined in the same step. If a clause is satisfied by this assignment, a final state will be reached. If not, an error state will be reached.

Now, construct $M_\varepsilon(\phi)$ from m^2 copies $M_1, \ldots, M_{m^2}$ of M_ϕ, such that the initial state of $M_\varepsilon(\phi)$ is the initial state of M_1, the initial state of M_{i+1} is the final state T of M_i, and reward 1 is gained if the final state of M_{m^2} is reached. The error states of all the M_is are identified as a unique sink state F.

To illustrate the construction, in Figure 3.1 we give an example POMDP consisting of a chain of 4 copies of $M(\phi)$ obtained for the formula $\phi = (\neg x_1 \vee x_3 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee x_4)$. The dashed arrows indicate a transition with probability $\frac{1}{2}$. The dotted (resp. solid) arrows are probability 1 transitions on action 0 (resp. 1). The actions correspond to assignments to the variables.

If ϕ is satisfiable, then a time-dependent policy simulating m^2 repetitions of any satisfying assignment has performance 1. If ϕ is not satisfiable, then under any assignment at least one of the m clauses of ϕ is not satisfied. Hence, the probability that under any time-dependent policy the final state T of $M(\phi)$ is reached is at most $1 - \frac{1}{m}$. Consequently, the probability

that the final state of $M_\varepsilon(\phi)$ is reached is at most $(1-\frac{1}{m})^{m^2} \leq e^{-m}$. This probability equals the expected reward. Since for large enough m it holds that $e^{-m} < (1-\varepsilon)$, the theorem follows. □

Unobservability is a special case of partial observability. Hence, we get the same nonapproximability result for POMDPs, even for unrestricted rewards.

Corollary 3.2.11. *Let $0 \leq \varepsilon < 1$. Any optimal time-dependent policy for POMDPs is ε-approximable if and only if* P = NP.

Corollary 3.2.12. *Let $0 \leq \varepsilon < 1$. The time-dependent value of POMDPs is ε-approximable if and only if* P = NP.

Note that the proof of Theorem 3.2.10 assumed a total expected reward criterion. The discounted reward criterion is also useful in the finite horizon. To show the result for a discounted reward criterion, we only need to change the reward in the proof of Theorem 3.2.10 as follows: Multiply the final reward by $\beta^{-m^2(n+1)}$, where β is the discount factor, m the number of clauses, and n the number of variables of the formula ϕ.

3.2.3 History-Dependent Policies

Although time-dependent policies are history-dependent, significant complexity can be added when the policy is given the full history of the computation rather than the time. In particular, the number of possible histories grows exponentially in the horizon. This is an intuition for the complexity of working with (or even storing) such policies.

Theorem 3.2.13. *The history-dependent finite-horizon policy existence problem for POMDPs is* PSPACE*-complete.*

The proof of Theorem 3.2.13 is a straightforward modification of the proof by Papadimitriou and Tsitsiklis (Papadimitriou & Tsitsiklis, 1987, Theorem 6), where the rewards for reaching the satisfying resp. unsatisfying final state are changed appropriately.

Theorem 3.2.14. *Let $0 \leq \varepsilon < 1$. The history-dependent finite-horizon value of POMDPs with non-negative rewards is ε-approximable if and only if* P = PSPACE.

Proof. The history-dependent value of a POMDP M can be calculated using binary search over the history-dependent policy existence problem. The number of bits to be calculated is polynomial in the size of M. Therefore, by Theorem 3.2.13, this calculation can be

performed in polynomial time using a PSPACE oracle. If P = PSPACE, it follows that the history-dependent value of a POMDP M can be *exactly* calculated in polynomial time.

The set QSAT of true quantified Boolean formulae is one of the standard PSPACE complete sets. To conclude P = PSPACE from an ε-approximation of the history-dependent value problem, we use a transformation of instances of QSAT to POMDPs similar to the proof of Theorem 3.2.13 in (Mundhenk, 2000).

The set QSAT can be interpreted as a two-player game: Player 1 sets the existentially quantified variables, and player 2 sets the universally quantified variables. Player 1 wins if the alternating choices determine a satisfying assignment to the formula, and player 2 wins if the determined assignment is not satisfying. A formula is in QSAT if and only if player 1 has a winning strategy. This means player 1 has a response to every choice of player 2, so that in the end the formula will be satisfied.

The version where player 2 makes random choices and player 1's goal is to win with probability $> \frac{1}{2}$ corresponds to SSAT (*stochastic satisfiability*), which is also PSPACE complete. The instances of SSAT are formulas which are quantified alternatingly with existential quantifiers $\exists$ and random quantifiers R. The meaning of the random quantifier R is that an assignment to the respective variable is chosen uniformly at random from $\{0, 1\}$. A stochastic Boolean formula

$$\Phi = \exists x_1 R x_2 \exists x_3 R x_4 \ldots \phi$$

is in SSAT if and only if

there exists b_1 for random b_2 exists b_3 for random b_4 $\ldots Prob[\phi(b_1, \ldots, b_n) \text{ is true}] > \frac{1}{2}$.

If Φ has r random quantifiers, then the strategy of player 1 determines a set of 2^r assignments to ϕ. The term "$Prob[\phi(b_1, \ldots, b_n) \text{ is true}] > \frac{1}{2}$" means that more than 2^{r-1} of these 2^r assignments satisfy ϕ.

From the proof of IP = PSPACE by Shamir (1992) it follows that for every PSPACE set A and constant $c \geq 1$ there is a polynomial-time reduction f from A to SSAT such that for every instance x and formula $f(x) = \exists x_1 R x_2 \ldots \phi_x$ the following holds.

- If $x \in A$, then $\exists b_1$ for random $b_2 \ldots Prob[\phi_x(b_1, \ldots, b_n) \text{ is true}] > (1 - 2^{-c})$, and
- if $x \notin A$, then $\forall b_1$ for random $b_2 \ldots Prob[\phi_x(b_1, \ldots, b_n) \text{ is true}] < 2^{-c}$.

This means that player 1 either has a strategy under which she wins with very high probability, or the probability of winning (under any strategy) is very small. We show how to transform a stochastic Boolean formula Φ into a POMDP with a large history-dependent value if player 1 has a winning strategy, and a much smaller value if player 2 wins.

For an instance $\Phi = \exists x_1 R x_2 \ldots \phi$ of SSAT, where Φ is a formula with n variables $x_1, \ldots, x_n$, we construct a POMDP $M(\Phi)$ as follows. The role of player 1 is taken by the controller of the process. A strategy of player 1 determines a policy of the controller, and vice versa. Player 2 appears as probabilistic transitions in the process. The process $M(\Phi)$ has three stages. The first stage consists of one step. The process chooses uniformly at random one of the variables and an assignment to it, and stores the variable and the assignment. More formally, from the initial state s_0, one of the states "$x_i = b$" ($1 \leq i \leq n$, $b \in \{0, 1\}$) is reached, each with probability $1/(2n)$. Which variable assignment was stored

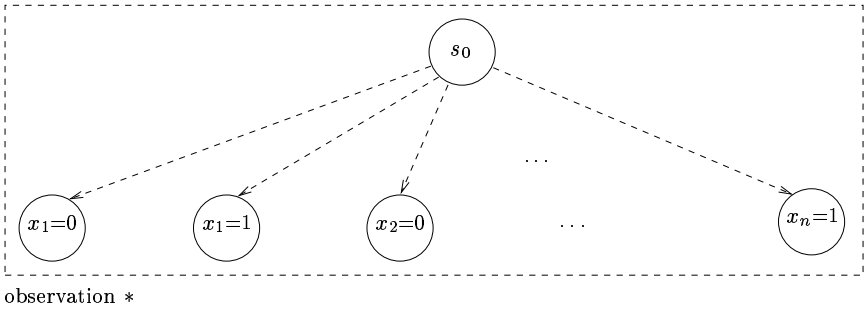


Figure 3.2: The first stage of $M(\Phi)$.

by the process is not observable. However, whenever that variable appears later, the process checks that the initially fixed assignment is chosen again. If the policy gives a different assignment during the second stage, the process halts with reward 0. (There is a deterministic transition to a final state which we refer to as s_{end}, or less formally, the *dead end* state.) If such an inconsistency occurs during the third stage, the process halts with reward 0 and notices that the policy *cheats*. (There is a deterministic transition to a sink state which we refer to as s_{cheat}, or less formally as *the penalty box* because the player sent there cannot re-enter the game later.) If eventually the whole formula is passed, either reward 2 or reward 0 is obtained dependent on whether the formula was satisfied or not. The first stage is sketched in Figure 3.2. In this and the following figures, dashed arrows represent random transitions (all of equal probability, regardless of the action chosen), solid arrows represent deterministic transitions corresponding to the action 1 (True), and dotted arrows represent deterministic transitions corresponding to the action 0 (False).

The second stage starts in each of the states "$x_i = b$" and has n steps, during which an assignment to the variables $x_1, x_2, \ldots, x_n$ is fixed. Let $A_{c,b}$ denote the part of the second stage of the process during which it is assumed that value b is assigned to variable x_c. If a variable x_i is existentially quantified, then the assignment is the action in $\{0, 1\}$ chosen by

the policy. If a variable x_i is randomly quantified, then the assignment is chosen uniformly at random by the process, independent of the action of the policy. In the second stage, which assignment was made to every variable is observable. If the variable assignment from the first stage does not coincide with the assignment made to that variable during the second stage, the trajectory on which that happens ends in the dead-end state that yields reward 0. Let r be the number of random quantifiers of Φ. Every strategy of player 1 determines

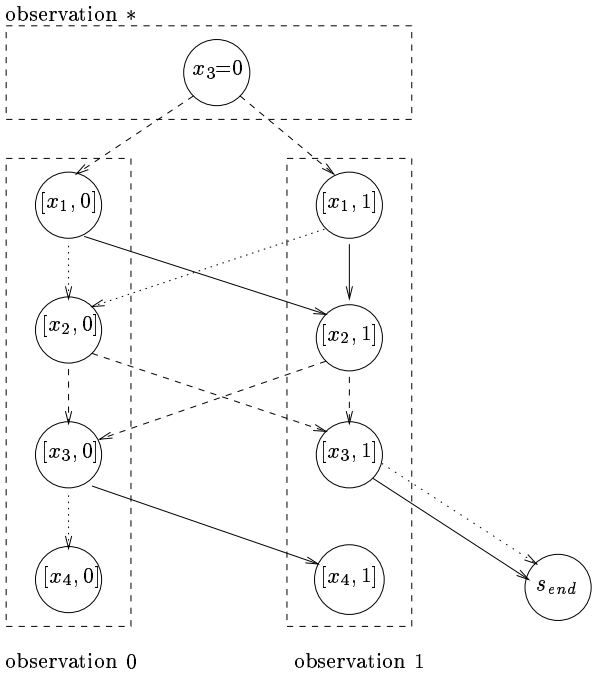


Figure 3.3: The second stage of $M(\Phi)$: $A_{3,0}$ for the quantifier prefix $Rx_1\exists x_2 Rx_3\exists x_4$.

2^r assignments. Every assignment $(x_1 = b_1, \ldots, x_n = b_n)$ induces $2n$ trajectories: n have the form

$$s_0, x_i{=}b_i, [x_1, b_1], \ldots, [x_i, b_i], \ldots, [x_n, b_n]$$

(for $i = 1, 2, \ldots, n$) that pass stage 2 without reaching the dead-end state and continue with the first state of the third stage, and n that dead-end in stage 2. The latter n trajectories that do not reach stage 3 are of the form

$$s_0, x_i{=}b_i, [x_1, b_1], \ldots, [x_i, 1 - b_i], s_{end}$$

(for $i = 1, 2, \ldots, n$). Accordingly, $M(\Phi)$ has $n \cdot 2^r$ trajectories that reach the third stage. The structure of stage 2 is sketched for $x_3 = 0$ in Figure 3.3.

The third stage checks whether ϕ is satisfied by that trajectory's assignment. The process passes sequentially through the whole formula checking each literal in each clause for an

assignment to the respective variable.[1] The case of a *cheating policy*, i.e., one that answers during the third stage with another assignment than fixed during the second stage, must be "punished". Whenever the variable corresponding to the initial, stored assignment appears, the process checks that the stored assignment is consistent with the current assignment. If eventually the whole formula passes the checking, either reward 2 or reward 0 is obtained, depending on whether the formula was satisfied and the policy was not cheating. Let $C_{c,b}$ be that instance of the third stage where it is checked whether x_c always gets assignment b. It is essentially the same deterministic process as defined in the proof of Theorem 3.2.1, but whenever an assignment to a literal containing x_c is asked for, if x_c does not get assignment b the process goes to state s_{cheat}. Otherwise, the process goes to state s_{end}. If the assignment chosen by the policy satisfied the formula, reward 2 is obtained; otherwise the reward is 0.

The overall structure of $M(\Phi)$ is sketched in Figure 3.4. Note that the dashed arrows represent random transitions (all of equal probability, regardless of the action chosen), solid arrows represent deterministic transitions corresponding to the action True, dotted arrows represent deterministic transitions corresponding to the action False, and dot-dash arrows represent transitions that are forced, whatever the choice of action.

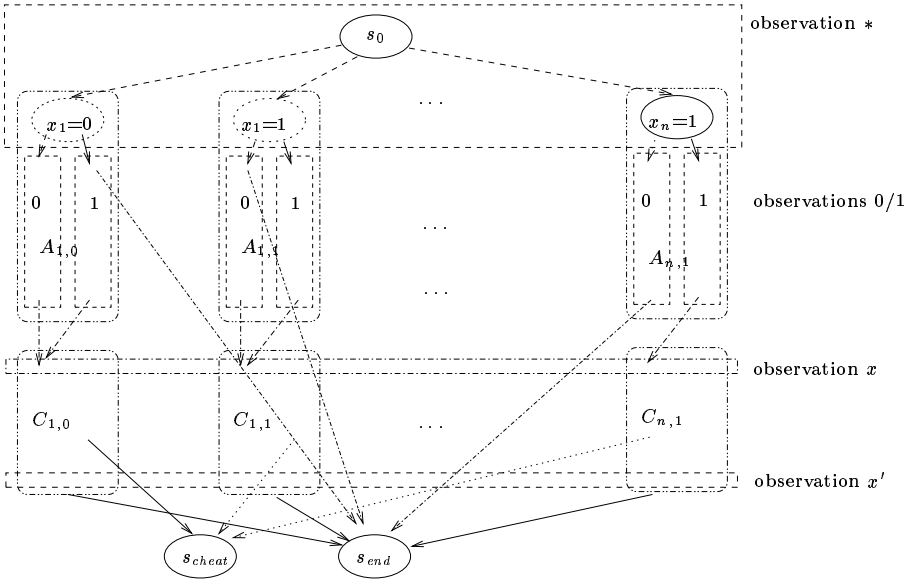


Figure 3.4: A sketch of $M(\Phi)$.

Consider a formula $\Phi \in$ SSAT with variables $x_1, x_2, \ldots, x_n$ and r random quantifiers, and consider $M(\Phi)$. Because the third stage is deterministic, the process has $2n \cdot 2^r$ trajectories,

[1] We can also regard the interaction between the process and the policy as an interactive proof system, where the policy presents a proof and the process checks its correctness.

$n \cdot 2^r$ of which reach stage 3. Now, assume that π is a policy, which is *consistent* with the observations from the n steps during the second stage, i.e. whenever it is "asked" to give an assignment to a variable (during the third stage), it does so according to the observations during the second stage, and therefore it assigns the same value to every appearance of a variable in $C_{k,a}$. Because $\Phi \in \text{SSAT}$, a fraction of more than $1 - 2^{-c}$ of the trajectories that reach stage 3 correspond to a satisfying assignment and are continued under this policy π to state s_{end} where they receive reward 2. Hence, the history-dependent value of $M(\Phi)$ is $> \frac{1}{2} \cdot (1 - 2^{-c}) \cdot 2 = 1 - 2^{-c}$.

For $\Phi \notin \text{SSAT}$, an inconsistent (or *cheating*) policy on $M(\Phi)$ may have performance greater than $1-2^{-c}$. Therefore, we have to perform a probability amplification as in the proof of Theorem 3.2.10 that punishes cheating. We construct $M_k(\Phi)$ from k copies $M_1, \ldots, M_k$ of $M(\Phi)$ (the exact value of k will be determined later), such that the initial state of $M_k(\Phi)$ is the initial state of M_1, and the initial state of M_{i+1} is the final state s_{end} of M_i. If in some repetition a trajectory is caught cheating, then it is sent to the "penalty box" and is not continued in the following repetitions. Hence, it cannot collect any more rewards. More formally, the states s_{cheat} of all the M_is are identified as a unique sink state of $M_k(\Phi)$.

If $\Phi \in \text{SSAT}$, then in each round (or repetition), expected rewards $> 1 - 2^{-c}$ can be collected, and hence the value of $M_k(\Phi)$ is $> k \cdot (1 - 2^{-c})$.

Consider a formula $\Phi \notin \text{SSAT}$. Then a non-cheating policy for $M_k(\Phi)$ has performance less than $k \cdot 2^{-c}$. Cheating policies may have better performances. We claim that for all k, the value of $M_k(\Phi)$ is at most $k \cdot 2^{-c} + 2n$. The proof is an induction on k. Consider $M_1(\Phi)$, which has the same value as $M(\Phi)$. Hence, the value of $M_1(\Phi)$ is at most 1. As an inductive hypothesis, let us assume that $M_k(\Phi)$ has value at most $k \cdot 2^{-c} + 2n$. In the inductive step, we consider $M_{k+1}(\Phi)$, i.e. $M(\Phi)$ followed by $M_k(\Phi)$. Assume that a policy π_j cheats on j of the 2^r assignments. From the n trajectories that correspond to an assignment, at least 1 is trapped for cheating under a cheating policy, and at most $n - 1$ may obtain reward 2. Then the reward obtained in the first round is at most $2^{-c} + 2 \cdot \frac{j \cdot (n-1)}{2n \cdot 2^r}$, and the rewards obtained in the following rounds are multiplied by $1 - \frac{j}{2n \cdot 2^r}$, because a fraction of $\frac{j}{2n \cdot 2^r}$ of the trajectories are sent to the penalty box. Using the induction hypothesis, we obtain the following upper

bound for the performance of $M_{k+1}(\Phi)$ under π_j for an arbitrary j.

$$\begin{aligned}
\mathrm{perf}_f(M_{k+1}(\Phi), \pi_j) &\leq \left(2^{-c} + \frac{j \cdot (n-1)}{n \cdot 2^r}\right) + \left(1 - \frac{j}{2n \cdot 2^r}\right) \cdot \mathrm{val}(M_k(\Phi)) \\
&\leq \left(2^{-c} + \frac{j \cdot (n-1)}{n \cdot 2^r}\right) + \left(1 - \frac{j}{2n \cdot 2^r}\right) \cdot \left(k \cdot 2^{-c} + 2n\right) \\
&= (k+1) \cdot 2^{-c} + 2n - \frac{j}{2^r} \cdot \left(\frac{1}{n} + \frac{k \cdot 2^{-c}}{2n}\right) \\
&\leq (k+1) \cdot 2^{-c} + 2n
\end{aligned}$$

This completes the induction step. Hence, we proved that, for $\Phi \notin \textsc{Ssat}$ and for every k, the value of $M_k(\Phi)$ is at most $k \cdot 2^{-c} + 2n$.

Eventually, we have to fix the constants. We choose c such that $2^c > \frac{\varepsilon-2}{\varepsilon-1}$. This guarantees that

$$(1-\varepsilon) \cdot (1 - 2^{-c}) - 2^{-c} > 0.$$

Next, we choose k such that

$$2n < k \cdot ((1-\varepsilon) \cdot (1 - 2^{-c}) - 2^{-c}).$$

Let $\widehat{M(\Phi)}$ be the POMDP that consists of k repetitions of $M(\Phi)$ as described above. Because k is linear in the number, n, of variables of Φ and hence linear in the length of Φ, Φ can be transformed to $\widehat{M(\Phi)}$ in polynomial time. The above estimates guarantee that

$$\text{value of } \widehat{M(\Phi)} \text{ for } \Phi \notin \textsc{Ssat} \quad \leq \quad k \cdot 2^{-c} + 2n \quad < \quad (1-\varepsilon) \cdot k \cdot (1 - 2^{-c}) \ .$$

The right-hand side of this inequality is a lower bound for an ε-approximation of the value of $\widehat{M(\Phi)}$ for $\Phi \in \textsc{Ssat}$. Hence,

- if $\Phi \in \textsc{Ssat}$, then $\widehat{M(\Phi)}$ has value $\geq k \cdot (1 - 2^{-c})$, and
- if $\Phi \notin \textsc{Ssat}$, then $\widehat{M(\Phi)}$ has value $< (1-\varepsilon) \cdot k \cdot (1 - 2^{-c})$.

Hence, a polynomial-time ε-approximation of the value of $\widehat{M(\Phi)}$ shows whether Φ is in $\textsc{Ssat}$.

Concluding, let A be any set in PSPACE. A polynomial-time function f exists which maps every instance x of A to a bounded error stochastic formula $f(x) = \Phi_x$ with error 2^{-c} and reduces A to $\textsc{Ssat}$. Transform Φ_x into the POMDP $\widehat{M(\Phi_x)}$. Using the ε-approximate value of $\widehat{M(\Phi_x)}$, one can answer "$\Phi_x \in \textsc{Ssat}$?" and hence $x \in A$ in polynomial time. This shows that A is in P, and consequently P = PSPACE. □

Corollary 3.2.15. *Let $0 \leq \varepsilon < 1$. The history-dependent value of POMDPs with general rewards is ε-approximable if and only if* P = PSPACE.

Note that it follows from Theorem 3.2.9 that the policy existence problem for history-dependent policies in the infinite horizon is undecidable. There is, however, a resource-bounded ε-approximation for the discounted reward metric: see Section 4.3.

3.3 Free Finite Table Policies

Remember that a free finite table (FFT) policy for a POMDP is a mapping from *observation* $\times$ *memory* to *action* $\times$ *memory*.

Theorem 3.3.1. *For any POMDP M and any $k \in \mathbb{N}$, there exists a POMDP M_k such that there is a straightforward isomorphism between the set of finite-table policies with k memory states for M and the set of stationary policies for M_k. Moreover, the corresponding policies have the same value.*

Proof. Given $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, t, r, o \rangle$ and $K = \{1, \ldots, k\}$ we construct M_k as follows.

- $\mathcal{S}_k = \mathcal{S} \times K$;
- $\mathcal{A}_k = \mathcal{A} \times K$;
- $\mathcal{O}_k = \mathcal{O} \times K$;
- $t_k(\langle s, i \rangle, \langle a, j \rangle, \langle s', l \rangle) = t(s, a, s')$ if $j = l$ and 0 otherwise;
- $r_k(\langle s, i \rangle, \langle a, j \rangle) = r(s, a)$;
- $o_k(\langle s, i \rangle) = \langle o(s), i \rangle$.

We refer to this as the *cross-POMDP*.

A policy π with k memory states for M translates directly into a stationary policy π_k for M_K by defining $\pi_k(\langle o, i \rangle) = \pi(o, i)$, and vice-versa. Observe that the rewards for following each policy for its respectively POMDP are the same. □

From this construction, it follows immediately that finding the optimal FFT policy for a POMDP is computationally equivalent to finding an optimal stationary policy. Therefore, by Theorem 3.3.1 and Theorem 3.2.1, we get immediate complexity analysis for FFT policies.

Theorem 3.3.2. *The FFT policy existence problem for POMDPs with nonnegative rewards over the finite horizon is* NP*-complete.*

All of the corollaries for Theorem 3.2.1 also follow for FFTs. For instance:

Corollary 3.3.3. *The FFT policy existence problem for POMDPs over the finite horizon is* NP*-complete.*

Thus, the problem is NP-complete for the infinite horizon under discounted rewards and NP-hard under average rewards, and NP-hard to ε-approximate in the finite horizon and the infinite horizon for discounted and average rewards.

3.4 Equivalence of Forms of Approximation for Partially Observable Markov Decision Processes

In the context of POMDPs, existence of a k-additive approximation algorithm and a PTAS are often equivalent. This might seem surprising to readers who are more familiar with reward criteria that have fixed upper and lower bounds on the performance of a solution, for example, the probability of reaching a goal state. In these cases, the fixed bounds on performance will give different results. However, we are addressing the case where there is no *a priori* upper bound on the performance of policies, even though there are computable upper bounds on the performance of a policy for each *instance.*

Theorem 3.4.1. *For POMDPs with non-negative rewards and flat representations under finite-horizon total or total discounted, and infinite-horizon total discounted reward metrics, if there exists a k-additive approximation then we can determine in polynomial time whether there is a policy with performance greater than 0.*

Proof. The theorem follows from two facts: (1) Given a POMDP M with value μ, we can construct another POMDP θM with value $\theta \cdot \mu$ just by multiplying all rewards in the former POMDP by θ; (2) under these reward metrics we can find a lower bound on μ if it is not 0.

The computation of the lower bound, δ, on the value of μ depends on the reward metric. Because there are no negative rewards, in order for the expected reward to be positive in the finite-horizon case, an action with positive reward must be taken with nonzero probability by the last step. Consider only reachable states of the POMDP, and let ν be the lowest nonzero transition probability to one of these states, h the horizon, and ζ the smallest nonzero reward, and set $\delta = \nu^h \zeta$. Then ν^h is a lower bound on the probability of actually reaching any particular state after h steps (if this probability is nonzero), in particular a state with reward ζ. If the reward metric is discounted, then let $\delta = (\gamma\nu)^h \zeta$, where $\gamma \in (0, 1]$ is the discount factor.

Now consider the infinite horizon under a stationary policy. This induces a Markov process, and the policy has nonzero reward if there is a nonzero probability path to a reward

node, i.e., a state from which there is a positive-reward action possible. This is true if and only if there is a nonzero-probability *simple* path (visiting each node at most once) to a reward node. Such a path accrues reward at least $\delta = (\gamma\nu)^{|S|}\zeta$ for stationary policies.

Since stationary policies have values bounded by the time-dependent and history-dependent values for infinite-horizon POMDPs, this lower bound for the stationary value of the POMDP is also a lower bound for other policies.

Finally, note that if the value of a POMDP with non-negative rewards is 0, then a k-additive approximation cannot return a positive value. To determine whether there is a policy with reward greater than 0 for a given POMDP, compute δ and then set θ such that $\theta\delta - k > 0$, i.e., $\theta > \frac{k}{\delta}$, and run the k-additive approximation algorithm on θM. The POMDP has positive value if and only if the approximation returns a positive value. □

Note that this does not contradict the undecidability result of Madani et al. (1999). The problem that they proved undecidable is whether a POMDP with nonpositive rewards has a history-dependent or time-dependent value of 0. We are asking whether it has value > 0 in the non-negative reward case; answering this question (even if we multiply the rewards by -1) does not answer their question.

Corollary 3.4.2. *For POMDPs with non-negative rewards, the POMDP value problem under finite-horizon or infinite-horizon total discounted reward is k-additive approximable if and only if there exists a PTAS for that POMDP value problem.*

Note that the corollary depends only on Facts (1) and (2) from the proof of Theorem 3.4.1. Thus, any optimization problem with those properties will have a k-additive approximation if and only if it has a PTAS.

Proof. Let $\mu = \mathrm{val}_\alpha(M)$, and let A be a polynomial-time k-additive approximation algorithm. First, the PTAS computes δ as in Theorem 3.4.1 and checks whether $\mu = 0$. If so, it outputs 0. Otherwise, given ε, it chooses θ such that $\theta > \frac{k}{\varepsilon\delta} \geq \frac{k}{\varepsilon\mu}$, and thus $\theta\mu - k > (1-\varepsilon)\theta\mu$ holds. Let $v = A(\theta M)$. (Note that $A(M)$ is the approximation to the value of M found by running algorithm A.) Then v is an ε-approximation to $\theta\mu$, so $\frac{v}{\theta}$ is an ε-approximation to μ.

Suppose, instead, that we have a PTAS for optimal policies for this problem. Let $A(M, \varepsilon)$ be an algorithm that demonstrates this. Let $\mu = \mathrm{val}_\alpha(M)$, and $A(M, 0.5) = v$. Thus $\mu \geq v \geq \frac{\mu}{2}$. If $\mu = 0$, then $v = 0$, and we can stop. Otherwise we choose an ε such that $(1-\varepsilon)\mu \geq \mu - k$, giving $\varepsilon \leq \frac{k}{\mu}$. Since $\frac{k}{2v} \leq \frac{k}{\mu}$, and $\frac{k}{2v}$ is polynomial size and is polynomial-time computable in $|M|$ (since v is the output of $A(M, 0.5)$), we can choose $\varepsilon < \frac{k}{2v}$, and run $A(M, \varepsilon)$. This gives a k-additive approximation. □

A problem that is not ε-approximable for some ε cannot have a PTAS. Therefore, any multiplicative nonapproximability result yields an additive nonapproximability result. However, an additive nonapproximability result only shows that there is no PTAS, although there might be an ε-approximation for some fixed ε.

Chapter 4
Historical Partially Observable Markov Decision Process Algorithms

This chapter provides an survey of work on Markov decision processes (MDPs) and partially observable Markov decision processes (POMDPs) prior to this thesis. Some of the techniques described have been used in the work described in Chapter 6, in particular, backward iteration (Section 4.2.1.2).

MDPs were originally formulated by Bellman (1957), who is responsible for most of the theoretical background of MDPs, the basic algorithms, and optimality results. POMDPs were coined by Sondik (1971) and his advisor Smallwood (Smallwood & Sondik, 1973) based on various work of Howard (1960), and Aoki (1965), and Astrom (1965). In the previous chapters we gave definitions, complexity results and motivating problems for MDPs and POMDPs. Thus we know that the problems are often hard and sometimes intractable, yet solutions to these problem would be desirable. In the following sections, we describe some possible solutions.

Three major problems are involved in using MDPs or POMDPs to solve an application: constructing the MDP or POMDP; finding a desirable policy for the controller; and evaluating a given policy for a given model. In the following section we will, as throughout the work, concentrate on the second and third problems. We will start with two fundamental concepts for most algorithms, and then we will discuss MDP techniques that form part of the foundation for all POMDP techniques; then we will discuss POMDP solving techniques, followed by POMDP heuristics.

In the case of POMDPs, because of the lack of efficient algorithms, one can consider two approaches: exact algorithms, which, in the limit, compute the optimal policy, and heuristics, which are guaranteed to halt, but are not guaranteed to find an optimal or even near-optimal policy.

More extensive and published surveys have appeared (Lovejoy, 1991b; White, 1991; Hauskrecht, 2000).

4.1 Value Functions and Belief Space

The keystones of MDP and POMDP solving and evaluation techniques are the value functions and the belief space for POMDPs. Most MDP/POMDP-solving algorithms rest on value functions, and most POMDP heuristics have a value function. The belief space is used

in many ways in heuristics and algorithms, for instance, to convert a POMDP into an MDP, for voting, or for making decisions.

A value function (V) is simply a mapping from states to expected rewards (see Section 2.3), i.e., the "value" of being in a giving state,

$$V : \mathcal{S} \rightarrow \mathbb{R}.$$

The most common value function is the one corresponding to the value of the optimal policy (V^*). In fact, for MDPs if one has V^*, then one also has the optimal policy π^*, in the infinite horizon, since

$$\pi^*(s) = max_{a \in \mathcal{A}} r(a, s) + \gamma(sum_{s' \in \mathcal{S}} t(s, a, s') V^*(s')).$$

In fact, most MDP algorithms don't find π^* but V^* and then induce π^* from it. One can take any value function V and use it to induce a policy π^V; however, if V is not close to V^*, then the performance of such a policy is not known.

Suppose one is an agent running a POMDP and wanting to decide what to do next. One of the facts one might look at is the probability that one is in a given state, based on the observations one has made and the actions one has taken. This probability distribution is a belief state (b). The set of all possible belief states is the belief space. Aoki (1965) showed that the belief state is sufficient information to capture all the history of a POMDP, i.e., that a policy that just looks at a correctly updated belief state to make decisions can be optimal (over all possible policies for the POMDP). The update method for the belief state is

$$b'(s') = \sum^{s \in \mathcal{S}} t(s, a, s') \cdot b(s) \cdot o(s', z),$$

where b is the old belief state, b' is the new belief state, a is the action chosen, and z is the observed observation. (Note that for deterministic observations, $b'(s') = 0$ if $o(s') \neq z$).

Frequently, when one is dealing with POMDPs, one represents the value function as a set of vectors over the belief space. To compute the value of a particular belief state, one chooses the vector which maximizes the dot product with that belief state. This vector representation is essential for several of the algorithms discussed below, in particular for value iteration.

4.2 Markov Decision Process Algorithms

In this section we will discuss various algorithms to find optimal policies or evaluate policies for MDPs. Much of this work was done by Bellman (1957) and is also covered in (Puterman, 1994).

Perhaps the most important part of Bellman's work is the proof of optimality for various policy types: For the infinite horizon under discounted-total-expected reward or average-expected reward we need only consider stationary policies, and for the finite horizon we need only consider time-dependent policies. From these proofs, Bellman gave algorithms for evaluating and finding optimal policies that will run in polynomial time in the size of the state space. Thus for an MDP with a finite state space, the problem can be considered solved in the theoretical sense, though numerical stability, constants, and the degree of the leading term in the polynomial may not be satisfactory to practitioners wishing to use these techniques.

We will start with techniques for the finite horizon (forward and backward iteration) and then progress to infinite-horizon techniques.

4.2.1 Forward and Backward Iteration

Forward and backward iteration are two primary methods for evaluating and finding the optimal policies for MDPs in the finite horizon. Backward iteration is also known as value iteration, since it computes the value function for the optimal policy of horizon length $i+1$ from the optimal policy for horizon length i. Backward iteration is also used in infinite-horizon cases.

In this section we give proofs of correctness and time bounds for forward and backward iteration algorithms.

4.2.1.1 Forward Iteration

Given a POMDP M, a stationary policy π, a starting distribution s_o and a horizon h, one can find the expected reward of π via dynamic programming in time $\Theta(|\mathcal{S}|^2 h)$. This algorithm is often called *forward iteration* in the literature (Puterman, 1994). The description of the algorithm is simplified by first proving the following theorem.

Theorem 4.2.1. *Given the expected reward and the expected distribution over the state space of M for a policy π at time t, one can in time $\Theta(|\mathcal{S}|^2)$ compute the expected reward and the expected distribution over the state space for time $t+1$.*

Proof. Define er_t, to be the expected reward at time t, and table ed_t to be the expected distribution at time t. So er_{t+1} is equal to er_t plus the likelihood we are in a state times the reward the controller will receive for the action chosen by the policy. Which is to say:

$$er_{t+1} = er_t + \sum_{s \in \mathcal{S}} ed_t[s] * r(s, \pi(o(s))).$$

This takes $\Theta(|\mathcal{S}|)$ operations.

Next, for each $s' \in \mathcal{S}$, $ed_{t+1}[s']$ can be computed in the same way by:

$$ed_{t+1}[s'] = \sum_{s \in \mathcal{S}} ed_t[s] * t(s, \pi(o(s)), s'),$$

which for each s takes $\Theta(|\mathcal{S}|)$ time. Such a calculation needs to be done for each $s \in \mathcal{S}$, so the whole computation of ed_{t+1} takes $\Theta(|\mathcal{S}|^2)$ time. This dominates the computation of er_{t+1}. Therefore, the whole update takes time $\Theta(|\mathcal{S}|^2)$, completing the proof of the theorem. □

Forward induction follows from the theorem. The expected reward at time 0 is 0, and the initial distribution (s_o) is the expected distribution. So one performs h updates as above and then returns er_h. This takes time $\Theta(|\mathcal{S}|^2 h)$.

With a modification to account for sparseness in the transition tables, the time bound drops though only to $O(|\mathcal{S}|^2 h)$. With the modification of replacing $\pi(o(s))$ with $\pi(o(s), t)$, forward induction works for the time-dependent case. The fully-observable case follows, since it is a special case of partial observability.

4.2.1.2 Backward Iteration

Backward iteration is another well-known algorithm. It is used for finding an optimal time-dependent policy (Puterman, 1994). It is also used in the infinite horizon, and for our work we use it as an evaluation for finite-memory policies. Backward iteration is based on the following theorem. This is an extension of Littman's (1994a) ideas to more general POMDPs.

Theorem 4.2.2. *If we can compute an optimal time-dependent policy for a horizon of length t and the expected reward for starting at each state, then we can compute the same for a horizon of $t+1$ in time $\Theta(|\mathcal{S}|^2|\mathcal{A}|)$.*

Proof. Define the table er_t to be the expected reward for each state at time $h-t$. Note that $\mathcal{O} = \mathcal{S}$ and $o(s) = s$ since we are in the fully observable case. For each $s \in \mathcal{S}$ we find $er_{t+1}[s]$ and $\pi((s, t+1))$.

Suppose that we are in state s with $t+1$ steps to go. Then the expected reward of action a is

$$r(s, a) + \sum_{s' \in \mathcal{S}} t(s, a, s') * er_t[s'],$$

i.e., the reward for the action plus the expected reward for the rest of time. This gives us

$$er_{t+1}[s] = max_{a \in \mathcal{A}}\{r(s, a) + \sum_{s' \in \mathcal{S}} t(s, a, s') * er_t[s']\}$$

in time $\Theta(|\mathcal{S}||\mathcal{A}|)$. We must do this for each $s \in \mathcal{S}$, giving us the time bound. Note that $\pi(s, t+1)$ becomes the a that maximizes the above expression. □

Backward iteration is simply h iterations of the above process with $er_1[s]$ set to 0 for all s. If expected reward er is required, it can be computed from er_h by

$$er = \sum_{s \in \mathcal{S}} er_h[s] * s_o(s),$$

where $s_0(s)$ is the likelihood of starting in state s. Note that for sparse transition tables, we can drop the bound from $\Theta(|\mathcal{S}|^2|\mathcal{A}|h)$ to $O(|\mathcal{S}|^2|\mathcal{A}|h)$.

4.2.2 Value Iteration

Now let us consider infinite horizon with total discounted reward. We can compute the value function over any finite horizon h just as we did above, with only minor modification to take into consideration the discount factor γ. The operator is:

$$V^h(s) = max_{a \in A} r(a, s) + \gamma(\sum_{s' \in \mathcal{S}} t(s, a, s') V^{h-1}(s')).$$

The astute observer will note that this is the backward iteration operator again, with minor modification for change in reward criteria. The value of the optimal policy is V^*, which is the fixed point of the operator when $0 < \gamma < 1$. In fact, if we have a stationary policy (the only interesting policies in the infinite horizon for Markov decison processes), we can also evaluate that policy since its value function will be the fixed point of the related operator

$$V_\pi^h(s) = r(\pi(s), s) + \gamma(\sum_{s' \in \mathcal{S}} t(s, \pi(s), s') V_\pi^{h-1}(s')).$$

In both cases, these operators do theoretically converge, with known rates. A problem to bring up is when $\gamma \sim 1$, is when the operators are most ill conditioned numerically and converge the slowest. Advantages of these methods are that they are easy to implement and are parallelisable.

Policy evaluation can be written as a system of linear equations. Note V_π is a vector over the real numbers. Let r_π be the vector of immediate rewards for policy π, and t_π be the transition matrix for π, i.e. $r_\pi(s) = r(\pi(s), s)$ and $t_\pi(s, s') = t(s, \pi(s), s')$. Thus one can write V_π as

$$V_\pi = r_\pi + \gamma t_\pi V_\pi$$

and so V_π is the solution to this system of linear equations:

$$(I - \gamma t_\pi)V_\pi = r_\pi \quad .$$

One can then use all the knowledge of how to solve systems of linear equations, in general and of this form.

The remaining question is how fast is value iteration, or more correctly, what is its rate of convergence. For policy evaluation, since this is just a system of linear equations, often sparse, one can use an iterative method like Jacobi iteration to solve the problem and get good convergence, provided the system is not too numerically unstable. One can also solve the system of equations exactly using Gauss-Jordan factoring, in time and memory polynomial in the number of states of the system. For finding the optimal policy, Tseng (1990) showed that a possibly exponential (in γ) number of iterations may be necessary for either an ε-approximation to the policy or for exactly finding the policy. However, in practice, value iteration often converges faster than this (Puterman, 1994).

4.2.3 Policy Iteration

Policy iteration is another method for finding the optimal policy for MDPs under the infinite horizon, in efforts to deal with the slow convergence of value iteration. The general idea is to start from a random policy and then move to better policies until the optimal one is reached. Each step of policy iteration costs about the same as each step of value iteration, so if policy iteration converges in fewer iterations, then we have a net gain in efficiency. Unfortunately, we don't get a theoretical gain in efficiency. Melekopoglou and Condon (1990) showed that a variation of policy iteration also can converge in an exponential number of iterations.

Policy iteration can be described as follows:

Pick a random stationary policy.

do

Compute V_π

for each $s \in \mathcal{S}$

$$\pi'(s) = \max_{a \in \mathcal{A}} r(s,a) + \gamma \sum_{s' \in \mathcal{S}} t(s,a,s')V_\pi(s)$$

Choose $\pi(s)$ when possible.

if $\pi'(s) \neq \pi(s)$ we have a better policy

$\pi = \pi$‘

while (π is changing)

Puterman (1994) gives convergence rates and proofs for policy iteration. The first argument is that policy iteration as stated above is equivalent to Newton's method for finding zeros on Blache operators. The second argument proves that if $\mathcal{S}$ is finite, and policy iteration and value iteration start at the same value function less than V^*, then the value function computed by policy iteration is bounded strictly between value iteration's value function and the optimal value function at each step.

Puterman also describes a third MDP algorithm, modified policy iteration, which is similar to both value and policy iteration. In fact, there is a parameter to the algorithm which, for different values, will give both value and policy iteration. The goal is that at each step of policy iteration, one does not often need to compute the value of the policy *exactly* to see how to improve it. Thus on large sparse MDPs, one can avoid computing the value exactly and hope to solve the MDP in a similar number of iterations compared to policy iteration. In practice, Puterman (1994, Section 6.7) shows for small MDPs policy iteration is faster, but larger MDPs have been solved with this method (Puterman & Shin, 1978; Thomas, Hartley, & Lavercombe, 1983).

4.2.4 Linear Programming

The last method we will describe is solving via linear programming. Here we describe the problem as a minimizing equation with constraints:

$$\text{minimize} \sum_{s \in \mathcal{S}} V(s)$$

subject to

$$V(s) - \sum_{s' \in \mathcal{S}} \gamma t(s, a, s') V(s) \geq r(s, a)$$

for all $a \in \mathcal{A}$ and $s \in \mathcal{S}$ which then one throws into one's favorite linear programming solver.

There are several problems with this method:

1. To our knowledge, there is no LP solving method that takes advantage of sparseness in the system; thus for large systems, this rapidly become infeasable;

2. LP problems are very sensitive to error; thus any error in the system could have detrimental effects, perhaps more than the other methods; and

3. The standard method that everyone learns (the simplex method) is numerically unstable and can take worst-case exponential time.

Thus, to use this method, one should purchase or acquire linear programming libraries from some source if one is not an expert in linear programming implementations. However, one can acquire more information from the linear programming solution than just optimal value function (e.g. the probablistic distribution over the state space for the optimal policy), so in some cases one might want to use the LP solution for this information.

4.3 Exact Partially Observable Markov Decision Process Algorithms

4.3.1 Constructing the co-MDP via Belief Space

In this section, we describe the underlying approach for most history-dependent policy algorithms and mention complexity analyses for some of these algorithms. Most algorithmic work on POMDPs uses history-dependent policies with discounted rewards over an infinite horizon, since the history-dependent policy is optimal and there are resource-bounded approximation schemes for finding it.

The algorithms are based around the belief state $b : \mathcal{S} \rightarrow [0, 1]$, a probability distribution over all the states of the POMDP. Let $\mathcal{B}$ be the set of all possible belief states. One then defines an MDP over the belief state called the co-MDP (Zhang & Liu, 1997a).

Unsurprisingly, it is not known how to compute V^* for the infinite horizon, so POMDP algorithms compute approximations to V^*. For any function $f : \mathcal{B} \rightarrow \mathbb{R}$, there is an induced policy

$$\pi^f(b) \triangleq \arg\max_{a \in \mathcal{A}} (r(b, a) + \gamma \sum_{z \in \mathcal{O}} \mu(z|b, a) f(b_+).$$

Value iterative methods use this policy, sometimes called the greedy policy, as their controller after producing an approximation to the optimal value function.

4.3.2 Value Iteration

One method of computing an approximation to V^* is value iteration. One starts with the zero function and computes successively better approximations to V^*.

Exact value iterative algorithms use V_t^*, the value function of the optimal policy with t steps left to go as their approximation. It is known (Bellman, 1957; Sondik, 1971) that if

$$\max_{b \in \mathcal{B}} |V_t^*(b) - V_{t-1}^*| < \varepsilon,$$

then the policy induced by V_t^*, is within $\frac{\gamma\varepsilon}{1-\gamma}$ of the optimal policy in performance. In other words,

$$\max_{b \in \mathcal{B}} \left| V^*(b) - V^{\pi^{V_t^*}(b)} \right| \leq \frac{\gamma\varepsilon}{1-\gamma}.$$

Sondik (1971) showed that V_t^* is representable by a finite set of vectors, and hence is piecewise linear and convex. Exact algorithms compute a minimal set of vectors to represent V_t^*.

The complexity of these algorithms comes from the number of vectors that they may need to represent the value function; this number grows exponentially with respect to t. If there are n vectors in V_t, then there are $|\mathcal{A}|n^{|\mathcal{O}|}$ candidate vectors in V_{t+1}. Experimental results from works like (Littman, 1996; Cassandra, 1998a) found that often little pruning occurs until t grows quite large. For further analysis of the complexity of exact algorithms, see (Littman, 1996; Cassandra, 1998a; Zhang & Liu, 1997b).

Note that, after sufficiently many iterations, we are guaranteed that the performance of the greedy policy will be within ε of the optimal performance, though this may require exponentially many iterations, each of which may take time exponential in the size of the POMDP. Currently, there is no algorithmic method for computing $\max_{b \in \mathcal{B}} |V_t^*(b) - V_{t-1}^*|$ unless there is an isomorphism between the vectors of V_t^* and V_{t-1}^*. Thus, if one depends on computing $\max_{b \in \mathcal{B}} |V_t^*(b) - V_{t-1}^*|$, there is no guarantee that this process will halt. One could, however, pre-compute a worst-case analysis on the number of iterations needed, and use that as a halting criterion.

4.3.3 Value Iteration via Incremental Pruning

There have been several algorithms for computing V_t^* from V_{t-1}^*, starting with Sondik (1971). The next algorithm developed was by Cheng (1988). Next was Lark III (1990), which had minor errors as stated, but those were corrected in his implementation and by Kaelbling, Littman, and Cassandra (1998). Currently, the best pure exact value iteration algorithm is Incremental Pruning by Zhang and Liu (1997a).

4.3.4 Acceleration of Incremental Pruning

Though pure Incremental Pruning is superior to previous value iterative methods, it still is not sufficient to solve all problems. In an effort to lessen the number of iterations required, Hansen (1997) implemented a new type of policy iteration. In his algorithm, one starts with a policy represented as a finite automaton with output. (For a description of this representation, see Section 4.6.) At each iteration, it computes a set of vectors representing the policy value, then performs a value iterative update. If the algorithm has not converged, it then updates the policy automaton, using the two sets of vectors.

Zhang and Zhang (2001) also uses a similar method but only worked with the value

functions. Unlike Hansen's work, which allowed any form of value iterative update, this work requires that either the Witness or Incremental Pruning algorithm is used. It attempts to improve the value function at the witness points: namely, those points that were used to witness that a vector belonged in the set of value vectors. Both the Witness algorithm and Incremental Pruning generate a list of such points. This is the current state of the art for exact value iteration algorithms.

4.4 Grid Methods

Exact value iteration methods are computationally intensive. Some of the heuristic methods to get around this problem include finite-history policies (White & Scherer, 1994; Platzman, 1977) and grid-based approximations(Drake, 1962; Lovejoy, 1991a). Hauskrecht (1997) showed that these two methods were equivalent, in that finite-history policies are a type of grid-based policy with a particular set of grid points (to be defined below).

In a grid-based method, one chooses a set of points in the belief space and then computes value iteration for those points until convergence of the Bellman residual at those points. This necessitates interpolating values for the rest of the belief space. The traditional grid-based method (Lovejoy, 1991a) evenly distributes the points across the belief space, including the corners, hence the name "grid". It has the advantage of having an easy interpolation function, but the disadvantage that the points are evenly distributed, and thus perhaps not concentrated in regions of high variation. The run time of a grid-based method is dependent on two factors: the number of points and the interpolation function.

The quality of grid-based methods is dependent on how close the points are to where they need to be. If a value function is represented by a single vector in a large region, one only needs a few points in that region. However, if the value function requires many vectors, the grid needs many points in that region to distinguish the different vectors. Each increase of precision in the regular grid algorithm requires a constant multiple increase in the number of points, leading to exponential growth in the number of points.

Several irregular grid-based methods have been explored (Hauskrecht, 1997; Brafman, 1997; Hauskrecht, 2000). One drawback to irregular grid methods, however, is interpolation. In order to interpolate the value at a point not on the grid, the "nearby" grid points must be found, and an interpolation function computed. In Zhou and Hansen (2001), a compromise is proposed: The resolution of the grid is set differently for each region of the belief space. Regions are defined by a low-resolution regular grid. According to (Zhou & Hansen, 2001), this is faster than Hauskrecht's and Brafman's algorithms, and so allows for more grid points,

and thus greater accuracy.

4.5 Pure Heuristics

All of the methods discussed so far in this chapter are computationally intensive. One might wish to avoid these methods for large examples and instead use some cheap method that has been known to work on other examples. Cassandra, Kaelbling, and Kurien (1996) and Cassandra (1998a) did an extensive study of such methods. Among the ones studied are:

- **Most likely state.** One computes an optimal policy for the underlying MDP, then at each time step, one performs the action corresponding to the MDP state with highest probability in the belief state.
- **Action voting.** As with most likely state, one computes the optimal policy for the underlying MDP, but at each time step one computes the probability, based on the belief state probabilities for each state, that a given action would be chosen; the one with the highest probability wins.
- **Minimizing entropy controls.** Instead of taking an action that maximizes some reward estimation, one takes an action that leads to the expected maximum drop in the entropy of the belief state. In other words, the policy tries to maximize the chance of unambiguously recognizing the state of the system at the next time step. (Entropy is computed by $H(b) = -\sum_{s \in \mathcal{S}} \log(b(s)) \cdot b(s)$.)
- **Dual mode controls.** An entropy-minimizing method is combined with an expected reward maximizing method; the choice of method at each time step depends on the entropy of the belief state.

These and others were compared on nine POMDPs, including four used in our work. The only clear winner in terms of performance in these comparisons was a grid-based method. For most of these methods, there are cases where they fail miserably. For instance, in most likely state, when the entropy of the belief state is high, it acts randomly. In minimizing entropy controls, expensive reset actions may be taken. One of Cassandra's examples was a model of a network; the best reset action available there was to reboot the entire network. This has obvious drawbacks as a regular choice of action.

4.6 Finite Automata and Vector Heuristics

The methods discussed in this chapter so far fall into two categories: those that can be evaluated, but are slow, and those that are faster, but cannot necessarily be evaluated. Frequently, a policy designer wants a policy that has a known value. A few more types of policies are evaluable. Here, we discuss one such type of policy, namely finite-automata-based methods.

The idea of representing a policy as a finite automaton grew out of the work of Sondik (1971), Cassandra et al. (1995), and Hansen (1998b, 1998a). A policy is represented as a finite automaton with output, where the states of the automaton correspond to a vector in a value function. Transition edges correspond to observations, and each state has an action output. (For an introduction to finite automata, see (Hopcroft & Ullman, 1980).) The advantages of the finite automaton representation are that one can explicitly evaluate the policy, and one can choose an arbitrary size of policy for which to search.

Hansen (1998a) first used this representation for accelerating value iteration and then proposed a heuristic search over the space of all finite automata policies. (A clearer description of the heuristic search is available in (Li, 1999).) The idea of searching over the space of fixed-size automaton policies was introduced in (Meuleau, Kim, Kaelbling, & Cassandra, 1999; Meuleau, Peshkin, Kim, & Kaelbling, 1999; Peshkin, Meuleau, & Kaelbling, 1999). They used several techniques to find deterministic and stochastic finite automata: exhaustive search over the deterministic automata, gradient-based local search and learning of stochastic automata. Later, they applied this approach to finding finite automata policies for factored MDPs (a more succinct representation of MDPs), as well (Kim, Dean, & Meuleau, 2000).

Chapter 5
Free-Finite-Table Policies

Until (Lusena, Li, Sittinger, Wells, & Goldsmith, 1999; Meuleau et al., 1999), there were only two methods of finding policies that were relatively fast and for which we could compute the performance: stationary policies and, in the finite horizon, time-dependent policies. However, there are examples where the optimal policy of either type performs quite badly. For instance, consider the POMDP shown in Figure 5. The solid lines represent deterministic transitions, independent of the action chosen. The dotted lines represent uniformly random transitions. Labels represent observations. There are two actions, neither of which affects the transitions. The only difference between the actions is on the reward at the states labeled "0". One action gives reward 1 in the left state, and reward -1 in the right state, and the other action reverses these rewards. Any stationary or time-dependent policy will be wrong with probability $\frac{1}{2}$ for the states labeled with observation 0, and thus has expected value 0, whereas the optimal history-dependent policy remembers the previous observation and thus, when it observes 0, can distinguish the two states. Therefore, it can choose the action with positive reward.

Free-finite-table (FFT) policies are both relatively easy to compute and to evaluate. In many cases, they are optimal or close to optimal. For instance, in the POMDP just described, a two-memory state policy is sufficient to be optimal.

FFT polices are similar to finite automata with output, described above, but do not explicitly represent value functions. There are, however, projections to and from value functions via finite automaton policies. We call our policies FFT policies because we place no restrictions on how the controller may use its memory, as opposed to other heuristic policy types such as time dependent or finite history (White & Scherer, 1994), and also because policies can be thought of as tables. Remember that in FFT policies, the controller has two parts: an M-state memory and a $\mathcal{O} \times M$ table of <action, memory> pairs. If the controller is in memory state a and makes observation b, the controller takes the action specified in the table at $\langle a, b \rangle$ and sets the memory to the given state. More formally,

$$\pi_{FFT} : \mathcal{O} \times \mathcal{M} \to \mathcal{A} \times \mathcal{M}.$$

FFT policies have advantages over most other policy types, with the possible exception of finite automata policies, with which they are mutually interconvertible.

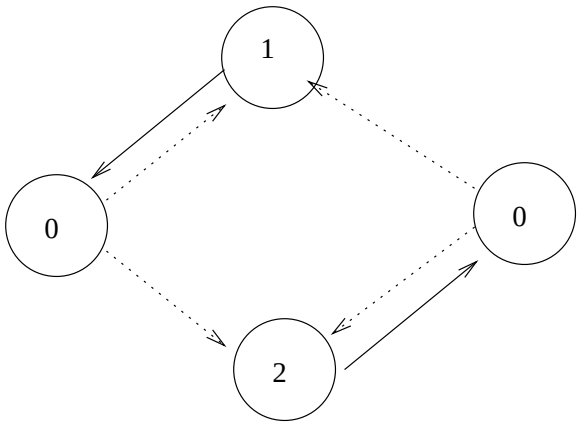


Figure 5.1: Time-dependent policies aren't always optimal

5.1 Comparing Free Finite Table Policies with Other Policy Types

5.1.1 Restricted Memory Policies

The earliest occurrences of finite memory policies all had explicit restrictions on what was remembered. The class of restricted memory policies includes stationary, time dependent, and finite-history dependent (White & Scherer, 1994; Platzman, 1977). These are policies when the agent remembers one of a **finite** set of facts, but the fact the agent remembers is restricted based on what is currently in memory and on the observation. Since they are all special cases of FFTs, the optimal FFT must always be at least as good as a restricted memory policy of the same size. Note that in some cases, a very small FFT may outperform a much larger restricted policy. Thus, FFTs are generally better than any of the restricted variations, except when there are far faster search algorithms for the restricted memory policies.

5.1.2 Pure Heuristics

The class of purely heuristic policies (Section 4.5) are policies that make decisions that are based on an *a priori* rule. These classes of policies can be and are used on very large systems because of their rule-based nature. (Thrun et al. (1999) describe a heuristically programmed robot that gave tours of one of the Smithsonian museums.) However, pure heuristics have several disadvantages:

1. They do not have a mathematical method of measuring performance. If one wishes to estimate their expected performance, one must either simulate the system or perform

experiments on the system running the policy; and

2. If the agent designer chooses a poor heuristic, the agent may perform extremely sub-optimally. For instance, if with a min-entropy rule, the agent always tries to maximize its certainty of its location and there is a very costly reset action, then the agent may continually reset the system for very poor performance. Thus great care must be taken in system design and rule choice.

Two major advantages of FFT policies over purely heuristic ones is that FFTs do not require prior domain knowledge to work, and they can be explicitly evaluated without being simulated or run.

5.1.3 Optimal Value Function Approximations

This is the class of policies where an approximation to the optimal value function is computed and then used to define a policy. There are several methods of computing approximations, including grid-based methods (Section 4.4), some finite automata methods, and others (Hauskrecht, 1997; Hauskrecht, 2000). Like FFTs, grid-based algorithms can be run on larger examples than the exact algorithms. Only some of the value function approximation policies can be evaluated mathematically. FFT methods appear to scale to larger examples better than value function approximation methods.

5.1.4 Exact Value Iteration

The exact methods don't scale. One problem here is that the exact solution in the infinite horizon is not computable (Madani et al., 1999). Another problem is that value iteration has a potential doubling of the number of value vectors at each step (Littman, 1996). This is the reason many people, including ourselves, look at heuristics for this problem.

5.2 Conversions Between Finite-Automata and Free-Finite-Table Polcies

There is a very close correspondence between FFT policies and finite automata policies, as shown by the following two theorems.

Theorem 5.2.1. *Every finite automata policy can be represented as an FFT policy.*

Proof. Given a finite automata policy π_a for a POMDP M, we construct an FFT policy π_f for M. Each state of the finite automaton becomes a memory state for π_f. The entry in the table corresponding to $\langle o, m \rangle$ is $\langle a, m' \rangle$, where a is the action associated with the state m

of the automaton and m' is the state in the automaton reached from m via the transition labeled o. The starting state of the automaton is associated with memory state 0.

The finite table policy produces the same actions as the finite automaton policy, given the same input. The size of the table is equal to the number of states of the automaton times the number of labels present on transitions, which is equal to the number of observations in M. □

Theorem 5.2.2. *Every FFT policy can be represented as a finite automaton.*

Proof. Given an FFT policy π_f, we construct a finite automaton representation π_a. Each entry in the table becomes a state in the finite automaton. An initial, no-action state is added. From this state, the states corresponding to memory state 0 are linked by the observations corresponding to those states. Thereafter, each state has transitions to all states from the m column of the table, with the corresponding observations as labels.

The size of the finite automaton is $\mathcal{O}(|\mathcal{O}| \cdot |\mathcal{M}|)$, which is roughly the size of the table. □

Note that the size of the corresponding cross-POMDP (see Thoerem 3.3.1) for the finite automaton is larger than for the FFT, because the construction with the automaton creates a state in the cross-POMDP for each state of the POMDP and each state of the automaton. This creates a factor of $|\mathcal{O}|$ more states than simply creating a state of the cross-POMDP from a state of the POMDP and a memory state. We therefore argue that FFTs are a more efficient representation of these policies.

Chapter 6
Design of Experiments

In this chapter, we describe the heuristics used in the experimental portion of our research. The outcomes of these experiments are described in Chapter 7.

As was shown in Chapter 3, finding optimal policies for POMDPs is hard, even if we limit our search space to finite memory policies. However, one still wants to find good policies, and to do so reasonably quickly. In this chapter, we describe the techniques we have explored that find reasonable finite memory policies for POMDPs.

We introduced free finite table policies as a representation of finite memory policies. Although for some POMDPs, finite memory policies are not the optimal policy, they perform well for many instances, and finding the best such policy is only an NP-hard problem. This allows us to take advantage of prior work on heuristics for NP-optimization problems. (See, for instance, Hochbaum (1997) for a survey of such work.)

In this chapter, we describe two techniques that we used for searching the space of free finite table policies. These techniques, local search and branch and bound, serve different functions. Local search should be fast, with approximately polynomial time bounds for most instances and most runs, but is not guaranteed to find an optimal policy. Branch and bound, on the other hand, frequently has long run times, but is guaranteed to find an optimal policy when it halts.

The major contribution of this work was to introduce to the POMDP community

1. the notion of *fixed* finite table policies, and

2. the idea of explicitly searching the space of policies.

While the notion of finite memory policies goes back to Sondik (1971), we were the first to consider the full generality of policies represented as tables and to explicitly fix the size of the policy. This explicit bound distinguished our work from Hansen (1998a), our closest immediate predecessor.

6.1 Local Search and Variations

Local search is a technique well-known at least since Biblical times (see Genesis 26:14-33). It is a heuristic method for finding an apparently optimal solution to an optimization problem. The basic ingredients of a local search are a neighborhood structure on the space of all solutions; a method of evaluating and comparing solutions; a method for choosing an initial

solution; and a method for choosing, at each pass of the algorithm, a new and better solution from among the neighbors of the current solution. In discrete spaces, it is usual that the size of the neighborhood is polynomial in the size of the problem, and the evaluation and choice can be done in time polynomial in the size of the problem.

The basic local search algorithm, given a solution, finds a new and better solution, referred to as an *improving neighbor* in this work. This process is repeated until no improving neighbor is found. This guarantees that the solution is at least a local optimum, though it is not necessarily optimal.

In considering FFTs, the search space is the set of all possible policy tables for the given POMDP and memory size. The most commonly considered neighborhood of a table T, in what follows, is the set of tables that differ from T in exactly one entry, or location. The structure of the table imposes an ordering on those locations. This, in conjunction with an ordering on possible values for each entry, imposes an ordering on the neighbors of each table. We follow this order when searching the neighbors, though searches do not necessarily begin with the first neighbor in the order.

The description given above of a basic local search algorithm does not by any means specify an algorithm. Many parameters must be determined in a full specification. We ran experiments on 14 variations of local search. The variational parameters were:

- weighting of the starting location for each pass;
- choice criterion for next policy;
- handling of irrelevant table locations;
- depth of paths considered in each pass; and
- number of retries allowed (for paths of depth greater than one).

We begin by describing each of the parameters, and then specify each of the 14 variants.

Remember that the policies considered here are specified by tables. The neighborhood of one table consists of all tables that differ from it by one entry.

The first variation we look at is the weighting of the probability that a given neighbor will be chosen first by the algorithm in each pass. A standard choice is to choose that neighbor uniformly at random, *Uniform weighting* of the starting place in the search space. However, during preliminary experiments, we noticed that, on larger examples near the end, large regions of the search space held policy choices that were locally optimal. In an attempt to avoid these regions, we weighted the starting location by how long it had been since we had

searched that particular choice, a *Nonuniform weighting* of the starting place in the search space.

The next design decision was which improving neighbor to consider next. One option, *first improvement*, is simply to pick the first improvement found. While this can cut down on the length of a pass, it may not perform as well as other variants. A second common choice, *best improvement*, is to choose uniformly at random among the best possible neighbors. In our case, since the neighborhoods considered are large and evaluation expensive, this could lead to long run times. As a compromise, *local best improvement* looked at choosing the best entry from the first table location we found that had an improving choice.

It is possible that some table entry locations will never be used in the running of a policy. This arises more frequently when more memory is allowed to the policy. Some of the variations considered ignored this possibility, and others identified irrelevant locations and then did not attempt to modify them. Note that one can determine reachability by means of a standard depth-first search of the policy states; this is a conservative method, in that all locations labeled as irrelevant are, but not all irrelevant locations are labeled as such. This method was implemented in the variations that weeded out irrelevant entries.

Local search has the problem of getting stuck on local optima. One method for getting unstuck from local optima is to increase the number of neighborhood steps one takes in each pass. This increases the size of the neighborhood, but sometimes decreases the number of non-optimal local optima. In particular, here we looked at modifying two or three locations in a table in one pass. We also tried one variant, *variable depth local search*, in which, within one pass, we first find an improving neighbor or, if none are found, choose a neighbor at random. This fixes one location in the table, which remains fixed for the duration of the search portion of this pass. Then we repeat the search-or-choose-at-random segment, given that some locations are fixed, until all locations are fixed. Finally, to complete the pass, we choose the best of all policies visited during that pass. Again, this variant makes for longer passes, but larger and less regular neighborhoods, so it can sometimes avoid suboptimal local optima.

In variable depth local search, because of the randomnized choices, one may make several attempts to get off a local optimum. The number of retries is thus a parameter choice.

The actual variants we considered were as follows:

1. first improvement, straight local search;
2. local best improvement, straight local search;
3. best improvement, straight local search;

4. first improvement, removing irrelevant locations;
5. local best improvement, removing irrelevant locations;
6. first improvement, nonuniform weighting of starting location;
7. local best improvement, nonuniform weighting of starting location;
8. local best improvement, removing irrelevant locations, nonuniform weighting of starting location;
9. first improvement, depth 2 search;
10. first improvement, depth 3 search;
11. first improvement, variable depth search;
12. local best improvement, depth 2 search, removing irrelevant locations, nonuniform weighting of starting location;
13. local best improvement, variable depth search, nonuniform weighting of starting location, removing irrelevant locations, three retries; and
14. local best improvement, variable depth search, uniform weighting of starting location, removing irrelevant locations, three retries.

6.2 Branch and Bound

Note that local search algorithms are heuristic, in that they are by no means guaranteed to find optimal solutions. On the other hand, for this work, it was useful to actually find optimal solutions. For this purpose, we implemented a version of a branch and bound algorithm. This gave us, for each POMDP/memory size pair, the optimal policy for those constraints.

Branch and bound is a heuristic approach that searches the space of policies. The state space is given a tree topology, and irrelevant branches of the tree are not searched. The state space here is the state of *partial policies*.

Definition 6.2.1. *A* **partial policy** *is a policy table where entries need not be specified.*

Note that the set of partial policies includes all fully specified policies. In the tree of partial policies, the root is the completely unspecified policy, and a child of one partial policy is that policy, with one more table entry specified. Note that, if identical policies are

identified, this is a directed acyclic graph (DAG). In the course of the branch and bound algorithm, the DAG is explored as a tree, but individual partial policies are not expanded more than once.

The first question in designing this algorithm was how to evaluate partial policies. For pruning purposes, the evaluation of a partial policy needed to be an upper bound on the value of all its complete descendents. In order to do this, we made the following observation: Every (partial) policy for a POMDP determines a (partial) policy for the underlying MDP, where the action chosen for an observation is applied to all states corresponding to that observation.

A partial policy for an MDP can be evaluated by finding the optimal completion. As discussed in Chapter 4, for the finite horizon, this involves finding the optimal time-dependent policy, and for the infinite horizon, this involves finding the optimal stationary policy. Both can be found in a straightforward manner using Value Iteration.

We then observed that the optimal completion of the MDP partial policy induced by a POMDP partial policy is at least as good as the optimal completion of that POMDP partial policy. Therefore, the value of the optimal completion of the MDP partial policy is an upper bound on the value of the optimal completion of that POMDP partial policy. This means that, in particular, it is an upper bound on the optimal completions of any extension of that POMDP partial policy, namely, of any of the descendent nodes of that partial policy. Details of this evaluation method are discussed in Subsection 6.3.2.

To do the branch and bound search, we kept track of the best complete policy found so far. When an upper bound was found that was below the value of that policy, the branch was pruned.

Two things affect the running time of branch and bound. The first is how fast one can evaluate the policies, and the second is how much of the search space gets pruned away. The amount of space that gets pruned away is affected in part by how one chooses to traverse the DAG. Policy evaluation will be discussed below; here we discuss the traversal.

An obvious first choice of traversal is to impose an ordering on the policies based on the ordering of the entries, and to traverse the tree in order. This was done by Littman (1994a), Meuleau et al. (1999), although in different search spaces. We tried this traversal and found that it did not scale to policies with more than four to eight table entries. We suspect that this is why the papers cited here did not consider larger policies. The underlying problem with the naive approach is that it may take many expansions and corresponding evaluations before one finally reaches a policy with high enough value to allow the algorithm to do significant pruning. Therefore, the algorithms implemented were aimed at finding

reasonably good policies more quickly.

Our first improvement on the naive algorithm was to order the entries for each table location according to their value approximations. This algorithm did not halt even for small examples. The next attempt was to choose the heuristic ordering proposed by Littman (1994a), based on ordering by expected reward for each state. (This is an ordering on locations, rather than entries into the locations.) This also did not yield acceptable run times. Several other heuristic orderings for locations were also tried, with notable lack of improvement on speed.

We then decided that a good ordering on locations should be dependent on the given partial policy. Dynamic heuristics were then tried. The most successful ordering heuristic required that each location entry be explored. Of course, this made each individual expansion extremely time-consuming, but led to much higher pruning than was previously achieved; this heuristic halted on significantly larger examples than previous attempts.

In the main program, we have a stack of partial policies that we have yet to expand, together with their values. In each step of the loop, we pop a partial policy off the stack and see whether (a) the value is still above the best complete policy we have seen, and (b) it is a complete policy.

If the policy value is not above the best value found, the policy is discarded. If it is a complete policy with a better value than previously known, we update the best value found variable. If neither is true, we expand the policy. The expansion function returns a list of child policies whose values are all better than the best found value and who all expand the previous policy by filling in the same location with different entries. We then push the children in ascending order of value onto the stack.

The expansion function first computes all reachable locations. Then it forms a priority queue of all reachable locations based on past performance of those locations. The algorithm then goes through the locations in order of priorities. For each location, we fill in each possible value and see whether or not it is prunable. If not, we put it on a list. If, at the end of this process, the list is empty or has exactly one member, then we short-circuit the search of locations, return that location and increase its priority for future searches. Note that, if the list is empty, this prunes the entire partial policy. Similarly, if the location has exactly one possible expansion, that is a forced choice for expanding the partial policy.

If the list has more than one possible entry, we use a heuristic function described below to choose the best list and to adjust priorities for future queues.

The first heuristic that we used looked at two properties:

1. the size of the list;

2. the variation among values of the partial policies on the list.

Lists without variation were given low preference, and that location's priority was lowered for future queues. For the remaining lists, a preference was assigned based on preferring smaller lists with greater variation. Variation was quantized by computing $(\max - \text{mean})/(\max - \min)$.

This prioritizing heuristic worked fairly well: more instances halted than with previous techniques. However, some moderate-sized examples still needed more than a week of machine time. A second prioritizing heuristic was then implemented. Since checkpointing had been deployed, the new heuristic was set loose on the ongoing computations.

The second heuristic looked at three properties of the lists of entries for each location:

1. the size of the list;

2. the variation among values of the partial policies on the list; and

3. whether the maximum value was lower than the parent's value.

Lists with the same maximum value as their parent were given low preference, and that location's priority was lowered for future queues. This heuristic greatly improved the run times of the ongoing examples, more of which halted. This was the last improvement done on the branch and bound program. All but the largest examples halted.

6.3 Implementation

Several important decisions were required in implementation, including the input file type and internal representation. We implemented most of our work on an upgraded turbo sparc5 170Mhz with 256M of RAM using `ecgs` as our primary compiler for C and C++.

A large collection of POMDPs is available on the Internet. Many of these are in a format described by Cassandra (1997–99). This format assumes the following model for a POMDP: there is a set of states, S, a set of actions, A, and a set of observations, O; there is an initial belief state, I; a transition table, T, gives the probabilities of going from one state to another, given an action; a reward table, R, maps all *from-state, action, to-state, observation* tuples to the set of real numbers; an observation table, Q, gives the probability for each observation, given a state and a previous action; and there is a discount factor, γ.

Cassandra's file format for a POMDP begins with a header listing the discount factor, the type of rewards used (reward versus cost), and the number of states, actions, and observations in the POMDP. Each of these values is easily read into any POMDP data structure. The

initial belief state is also a part of the file's header. It can be determined by the keyword uniform or a list of probabilities that can be read directly into the initial belief state table.

The remainder of the file describes various pieces of the transition, reward, and observation tables. Each line of the input starts with a T, R, or O (indicating an entry into the Transition, Reward, or Observation table, respectively), and is followed by a list of coordinates, indicating the position(s) in the table being set. Finally, a number, vector, or matrix of values is given, which are input into the positions described by the coordinates.

6.3.1 Parser

Parsing a file with Cassandra's grammar is straightforward[1]. Storing the tables in an efficient manner is more difficult. In his POMDP parser, Cassandra uses an intermediate matrix structure to store two-dimensional matrices. The transition table and observation table are then simply arrays of this intermediate matrix (one matrix per action, where each matrix is *from-state by to-state*). The matrix structure is an array of linked lists, one list per row of the matrix. Each entry of a list contains the column and value of the entry. The reward table (*action by from-state by to-state by observation*) uses a more complex representation, in which each action has a linked list of matrices, which in turn are single values, vectors, or sparse matrices, depending on how the entry appears in the file.

The use of linked lists in these data structures indicates linear time search for any input value, since it is possible for an entry to be specified more than once in the POMDP file, and only the last value should be considered. Additionally, if the final representation stores the tables in a style other than that used in the file (for instance, our transition table is *from-state by action by to-state*), then converting the intermediate table into the final table requires some form of reordering of the lists with respect to the new style.

To solve these problems, we determined that the best data structure for each table in the intermediate representation was an AVL tree (Adel'son-Vel'skii & Landis, 1962), since it allows for insertion and searching in log time and its entries can be traversed in order in linear time. We use four tables altogether: the initial belief state, the transition table, the reward table, and the observation table. These tables have dimensions one, three, four, and three, respectively. Our goal is to minimize memory usage, a significant bottleneck using a purely dense set of tables, while preserving the speed of the parser.

The data structure we use has the capability of representing a table in either a dense or a sparse format. Additionally, two different forms of sparsity are allowed. In one form,

[1]The parser used for these experiments was designed and implemented by Chris Wells in 1998–9, in collaboration with Christopher Lusena.

there is an entry for each non-zero entry of the initial table, while in the other, a single entry may represent successive equivalent non-zero entries. The data structure is named SoDIArray, short for "Sparse or Dense Input Array". To simplify the representation, the number of dimensions and each dimension's size are stored in the SoDIArray, and all indices in the table are converted to their linear index equivalent. Thus each entry requires only two numbers: the entry's value and the entry's linear index.

The SoDIArray structure has three substructures. For each input, two of those structures are always empty. It has a dynamic, linear array for the dense representation, an AVL tree for the first sparse representation, and an AVL tree for the second sparse representation. In the parser, each table starts out in the first sparse representation. As more entries are added to a table, it becomes possible that the current representation is less efficient than one of the other two representations. Conversion among the representations is needed, and a threshold of .9 was set (meaning a new representation has to be no larger than 90% of the current representation for a conversion to occur) in order to prevent thrashing. Most of the inputs we considered required no more than a single conversion with this threshold, and the conversion occurred early in the input. This means that a conversion will typically take time equal to the current number of non-zero entries, which is small relative to the total number of non-zero entries and very small relative to the size of the intermediate data structure at the end of parsing. With these three data structures, the time it takes to insert a new entry becomes $\mathcal{O}(\log n)$.

In the POMDP file format, however, several contiguous entries are often specified on a single line (for instance, *R : 0 : 0 : * : * 5*, meaning that for action 0 and from-state 0, any to-state/observation pair will yield a reward of 5). Instead of entering each indicated action/from-state/to-state/observation tuple separately, we allow the value, indices, and length of the entry to be input into the SoDIArray table. If the table happens to be using the second sparse representation, then the time it takes to input this range of entries becomes $\mathcal{O}(\log n)$ instead of $m\mathcal{O}(\log n)$, where m is the length of the new entry.

Once parsing is complete, every table but the reward table must be checked to ensure that all probabilities sum to 1. We provide a function for SoDIArrays called checkSum, which computes the sums for each row and returns an array with the sums for each row. For instance, calling checkSum(1) on the transition table would result in an array *action by states* large being returned, where each entry was the sum of the probabilities across an action/from-state pair. For sparse representations, this function takes $\mathcal{O}(n)$ time, where n is the number of non-zero elements, and the array is small relative to the table (unless we sum over a high dimension, which does not occur in the POMDP parser).

POMDP file	actual size (bytes)	dense size	% dense size used
McCallum's Maze	3072	33640	9
Sutton's Grid World	11680	1009672	1
Wilson's Woods	458112	1491827776	0
`aloha.30`	1654200	7580272	21
`aloha.10`	73360	266032	27
`4x3.95`	5872	29416	19

Table 6.1: Memory Usage for Dense Files

In developing this parser, we included memory metrics so we could determine the savings in memory we achieved. The results for several POMDP files are shown in Table 6.1.

Notice that the biggest savings occurred in the first three grid world examples. This is explained by a very sparse transition table (only four or eight possible to-states for any from-state) and a very sparse reward table (one or a few goal states).

The final step of parsing is to convert the intermediate structure into a useful representation. Our current work focuses on POMDPs with deterministic observations, so the reward table is reduced from *action by from-state by to-state by observation* to *from-state by action*. The transition table in our representation is *from-state by action by to-state*, and our observation table is a single array, in which each state is mapped to an observation.

We use a structure which again attempts to minimize memory use while preserving processing speed. We chose a row major sparse representation because of the nature of our calculations in our evaluator. In this representation, there are three arrays: an array with the start index of each row in the other two arrays, an array with the column number of each entry, and an array with the value of each entry. If a dense representation is smaller for a particular matrix, it is used instead. The structure for this matrix is called a SoDMatrix (for Sparse or Dense Matrix). The initial belief state and observation tables are both stored in dense format, because their sizes are linearly dependent on the number of states, and the observation table is always dense. The transition table is an array, state elements large, of SoDMatrix's, and the reward table is a single SoDMatrix.

The first step in our conversion is to determine whether or not the POMDP is truly deterministically observable. We first check the intermediate observation table for entries that are not zero or one. This involves checking each entry in a dense representation, or every non-zero entry in one of the two possible sparse representations. Then we check the intermediate reward table for action/from-state pairs that do not determine a single

reward. Again, in a dense representation, every entry must be examined. In the sparse representations, only every non-zero entry of the table must be checked.

Once the POMDP has been checked, the intermediate tables must be converted to the final tables. The conversion of the initial belief state is straight forward. Each non-zero entry is copied into the final dense table. The observation table's conversion is also simple. Each non-zero entry (at this point guaranteed to be 1.0), has indices *action by state by observation.* Each non-zero entry's observation index is stored in the state position of the final observation table. Both of these conversions are bounded by the number of states.

Converting the transition table is more complicated. We first calculate the number of non-zero entries for each from-state, which is $\mathcal{O}(n)$. Then an array of SoDMatrix's is declared, number of states large, and each SoDMatrix is marked as being either sparse or dense, depending on which one will store the indicated number of elements more compactly. If the intermediate table is dense, we simply copy each non-zero element from it to the new transition table. If the intermediate table is sparse, we can still efficiently copy its non-zero entries to the new intermediate table, because even though we are reversing the ordering of actions and from-states, the ordering of actions and to-states is preserved. This means that, for any given from-state, all action 0 entries will be examined before any action 1 entries occur, which allows us to fill in our row major arrays from left to right.

Finally, the reward table must be converted to a single SoDMatrix, states by actions in size. Recall that at this point in the conversion, we know that a from-state/action pair determines a single reward. So the number of non-zero elements in the intermediate table can be divided by number of states times number of observations to yield the number of non-zero elements in the final table, allowing us to easily determine the best representation for the reward table. If that representation is dense, every non-zero value for each from-state/action pair is copied into the dense array. If the representation is row-major, we must first determine how many non-zero entries occur for each from-state. As previously described, this can be done in $\mathcal{O}(n)$ time, and the results returned in an array of size number of states. Then we examine each non-zero entry in the intermediate table and insert it into the appropriate location in the final table.

The memory usage results for the final representation are shown in Table 6.2.

Because our final representation is determined by its usefulness in our calculations as well as by memory usage, the savings in memory are not as large as for the intermediate representation. Thus, for our work, the intermediate representation removed a critical bottleneck to the size of the POMDP we could parse.

The total time it takes to read in each POMDP file and convert it to the final repre-

POMDP file	actual size (bytes)	dense size	% dense size used
McCallum's Maze	1494	4670	31
Sutton's Grid World	5598	74006	7
Wilson's Woods	174216	60316568	0
`aloha.30`	708220	1903528	37
`aloha.10`	46168	68128	67
`4x3.95`	2798	4670	59

Table 6.2: Memory Usage Results

sentation is, on a human scale, very small for every example except `aloha.30`, which took several seconds. Its relative slowness may be explained by a transition table that cannot be represented efficiently by the second form of sparsity and by the size of the input file.

6.3.2 Partial Policy Evaluation

A policy evaluator is a function from POMDP/policy pairs that outputs the value of that policy for that POMDP. Because we construct guaranteed optimal policies using branch and bound, we also need to evaluate partial policies.

Given a POMDP and a partial policy, we construct a (fully observable) Markov decision process (MDP) that shares states and actions and some transitions with the POMDP; those actions that are already specified by the partial policy are hard-wired into the MDP, and all other transitions are as in the POMDP. Thus we have reduced the problem to finding the value of the optimal policy for an MDP, a well-understood problem.

For finite-horizon evaluation of MDPs, one usually applies a value iterative technique using a form of dynamic programming called backward induction (Puterman, 1994). For the infinite-horizon case, one can also do this. There are also evaluation algorithms that depend on reducing the evaluation to a linear-programming problem. Other techniques are based on policy iteration.

We use value iteration, i.e., backward induction. One step of the iteration consists of taking the values of each state-memory pair at time $-t$ and calculating the values at time $-(t+1)$. [2]

One potential pitfall of backward induction is excessive use of memory. For instance, if we explicitly constructed the MDP for a finite-memory policy, instead of the standard *state*

[2]Time is counted backwards from the horizon; in an infinite-horizon calculation, we stop when the change in value from one epoch to the next is sufficiently small.

by state transition matrix, we would have a *state by memory by state by memory* matrix. Furthermore, this matrix would be regular and often sparse — and huge. We avoid this by constructing the MDP implicitly. To do so, we represent the vectors needed by backward iteration by matrices. (As an added benefit, we use BLAS3 operations in some cases instead of just BLAS1 and BLAS2.) Before the first step of the induction, the algorithm computes a minimal set of BLAS calls it will need, and reuses as much computation as possible for each step.

The finite-horizon evaluator runs for H steps, where the horizon, H, is an input parameter. The stopping condition for the evaluator in the infinite-horizon case is a parameter, ε, in the problem. The error at time $-t$ is the maximum change for any state-memory at time $-t$. We check when the error divided by the maximum value of any state-memory pair at that time is less than ε. For the infinite-horizon experiments reported here, we used $\varepsilon = 10^{-10}$.

6.3.3 Infinite Horizon Partially Observable Markov Decision Process Policy Evaluation

The bottleneck in all of the local search methods is policy evaluation. One can greatly speed up all methods by improving the speed of policy evaluations. Thus, we implemented and tested several different algorithms. All of the evaluation techniques used for the infinite-horizon runs were based on Value Iteration, as described in Chapter 4. Each of these techniques starts with an initial approximation to the value of the policy, and updates that approximation until the process converges. The differences are in the update methods and choice of initial approximation.

The two techniques for the update method are called Jacobi and Gauss-Seidel methods. Both are similar to the methods of solving linear equations by the same names. These are described in more depth in Chapter 4.

The two choices we used for the initial approximation were the zero vector and the value vector of the previous policy considered. The latter could be expected to provide a reasonable approximation most of the time, since the policy would have only changed in one location.

To test the different evaluators, several experiments were run. Each experiment consisted of one local search variant, one POMDP instance, and two evaluation methods. Within the experiment, every time a policy was evaluated, both methods were called and their outputs and running times compared.

Unfortunately, all four evaluators are numerically unstable. The evaluations, and thus the local searches, did not necessarily halt for the infinite-horizon cases. For `tiger.95` and `LocalSearch1`, each of the four evaluation methods ran into numerical instabilities that

caused nonconvergence.

One can, however, say something about run times when the methods do converge. As one would expect, the Gauss-Seidel method converges in about half the iterations of the Jacobi methods, and choosing the previous vector often leads to quick convergence. The outputs, when both methods halted, were consistently very close (usually within 10^{-13} relative error).

Chapter 7
Experiment Results and Analysis

In this chapter, we describe the actual experiments and their outcomes. We begin by describing the Partially Observable Markov Decision Processes (POMDPs) used for these experiments in Section 7.1. We list the actual experiments and give summaries of the performances of many of the experiments in Section 7.2. We discuss the outcomes of these experiments in Section 7.2, and make recommendations to future POMDP algorithm designers in Section 7.3.

7.1 Partially Observable Markov Decision Processes Instances

Most of the POMDPs used in this project were downloaded from the web site developed by Cassandra (1997–99). Some were entered from the literature: Wilson's Woods and Sutton's grid world are from Littman (1994a); Mazes 7, 10, and 20 are from Hauskrecht (1997).

The POMDPs are briefly described below, including information on their sizes. The Aloha Project was a packet network over radio waves that was a precursor to the ethernet. Several of the POMDPs used model the decisions for the network.

- `stand.tiger.95` models the game of lady and the tiger from (Stockton, 1882): The agent starts in a chair from where he can hear with some probability of error which door the tiger is behind when it roars. After he stands up (and increases the error probability), he chooses a door and get a reward if the "lady" is there and a penalty if the "tiger" is. *4 states, 4 actions, 4 probabilistic observations*

- `web.ad` models a web site trying to do targeted advertising. *4 states, 3 actions, 5 probabilistic observations*

- `network` models demands of network users, with an agent trying to control a network router. *6 states, 4 actions, 2 probabilistic observations*

- `aloha.10` models an aloha network with 10 nodes. *30 states, 9 actions, 3 observations*

- `1d` models navigation in a one-dimensional maze where all the non-goal states look alike. *4 states, 2 actions, 2 observations*

- `1d.noisy` is a variation of `1d` with nondeterministic actions. *4 states, 2 actions, 2 observations*

- `4x3.95` is a maze problem with nondeterministic actions. The name suggests the size of the maze. (It suggests 12 states, but one location in the maze is a brick.) *11 states, 4 actions, 6 observations*

- `4x5x2.95` is a larger maze problem with nondeterministic actions. *39 states, 4 actions, 4 observations*

- `maze7, maze10, maze20, sutton` are maze problems with deterministic actions and observations. *10, 19, 20, 47 states (respectively), 8, 8, 6, 4 actions, 8, 10, 8, 13 observations*

The following are significantly larger examples.

- `aloha.30` models an aloha network with 30 nodes. *90 states, 27 actions, 3 observations*

- `machine` models a machine maintenance example. *256 states, 4 actions, 16 observations*

- `iff` models an aircraft identification system (is this model of aircraft friend or foe?). *104 states, 4 actions, 22 observations*

- `wilson` Wilson's Woods is a maze with approximately 1000 states, including multiple goal states. *970 states, 8 actions, 95 observations*

7.2 Experiments and Graphical Summaries

For each of the small POMDPs, we ran all 14 variations of local search 100 times. For the larger POMDPs, we ran seven variations (numbers 1, 2, 4, 6, 8, 11, and 14 in the list in Chapter 6) 10 times each. This smaller number was because some of the runs took more than six hours of machine time. The goal was to run each variation considered on each POMDP for up to four memory states. In the slowest cases, this was not always feasible; in the cases of particularly small POMDPs, larger numbers of memory states were considered.

The number of memory states considered for each POMDP is listed below.

- `stand.tiger.95` was run with 1–10 memory states;

- `web.ad` was run with 1–10 memory states;

- `network` 1–4 memory states;

- `aloha.10` 1–4 memory states;

- `1d` 1–10 memory states;
- `1d.noisy` 1–10 memory states;
- `4x3.95` 1–4 memory states;
- `4x5x2.95` 1–8 memory states;
- `maze7` 1–4 memory states;
- `maze10` 1–6 memory states;
- `maze20` 1–3 memory states;
- `sutton` 1–4 memory states;
- `aloha.30` 1–4 memory states;
- `machine` 1–2 memory states;
- `iff` 1–4 memory states; and
- `wilson` 1–3 memory states.

Branch and bound was started on all of the POMDPs. It halted on all of the small POMDPs with 4 or fewer memory states, and a few larger instances or higher numbers of memory states, though the larger cases tended to take at least a week of machine time.

In the following subsections, we describe and illustrate conclusions drawn from our experiments.

7.2.1 More Memory Helps

Unsurprisingly, we found that adding policy memory increased the average values of the found policies for all local search variants and all POMDPs. As we observed in (Lusena et al., 1999), the most striking result of our research is that searches run with extra memory, i.e., more memory than is needed to get an optimal policy, are still more likely to actually *find* an optimal policy than searches that are constrained to policies with no "extra" memory. For instance, the experiments described in that paper considered McCallum's maze, which has an optimal policy with only two memory states, yet we saw that the optimal value was found more often when the search algorithms were run with more memory.

One explanation of this phenomenon is geometric: If there is a local optimum between the starting policy and the global optimum and only one path from the start to the global

optimum, a local search will get stuck at the local optimum. However, the extra memory functions as an additional dimension, allowing multiple paths from the initial policy to the global optimum. (Imagine trying to do hill climbing on the curve $x^3 - x$ from the point $(-1, 0)$: In two dimensions, one gets stuck on the hump at $x = -1/3$, but in three dimensions one might get a surface which does not peak at that x value for all z values, so one might be able to work around that local hump.)

The flip side of this discovery is that adding memory slows down computation, since it increases the size of the policy space. Therefore, one is faced with the time/quality trade-off: Allowing the search algorithm to search a larger space of policies dramatically increases the probability of finding an optimal policy, but also increases the expected time to convergence.

This effect was consistent for all the POMDPs considered and for all local search algorithms. We demonstrate it here for `web-ad`, `sutton`, and `aloha.30` in Figures 7.2.1 to 7.2.1. These POMDPs are chosen to represent the small, medium-sized, and large examples respectively. For the first two, we show local searches 1, 2, 8 and 11, but for `aloha.30`, some of these variants were too slow (see Sections 7.2.2, 7.2.3 and 7.2.4). For `aloha.30`, we show variants 1, 2, and 8. Lighter regions of the pie chart indicate policies with higher value. The color-to-value correspondence is consistent within each figure, but not necessarily across the figures.

7.2.2 Small Fixed-Depth Searches Are Worse Than Vanilla Search

We had originally conjectured that small expansions of the search space would buy at least small improvements in the performance of local search. What we discovered was that depth 2 and depth 3 searches performed no better than basic, unimproved local search, although it took significantly longer to get the same results. The timing data was consist, though sometimes depth 2 and depth 3 searches actually performed worse than basic local search.

Performance data comparing plain local search (variant 1) with depth 2, depth 3, and depth 2 with acceleration (variant 12) is shown in Figures 7.2.2 and 7.2.2. These versions of local search were not run on the large POMDPs, since they were too slow. These show the performances on `1d` and `sutton`.

Timing data comparing plain local search (variant 1) with depth 2, depth 3, and depth 2 with acceleration (variant 12) is shown in Figures 7.2.2 and 7.2.2. In all of the time bar graphs, each cluster of bars represents a local search variant, and each bar represents a memory size, from smallest/darkest to largest/lightest.

Figure 7.1: Performance of `web-ad` local searches 1, 3, 8, and 11 with 1-10 memory states

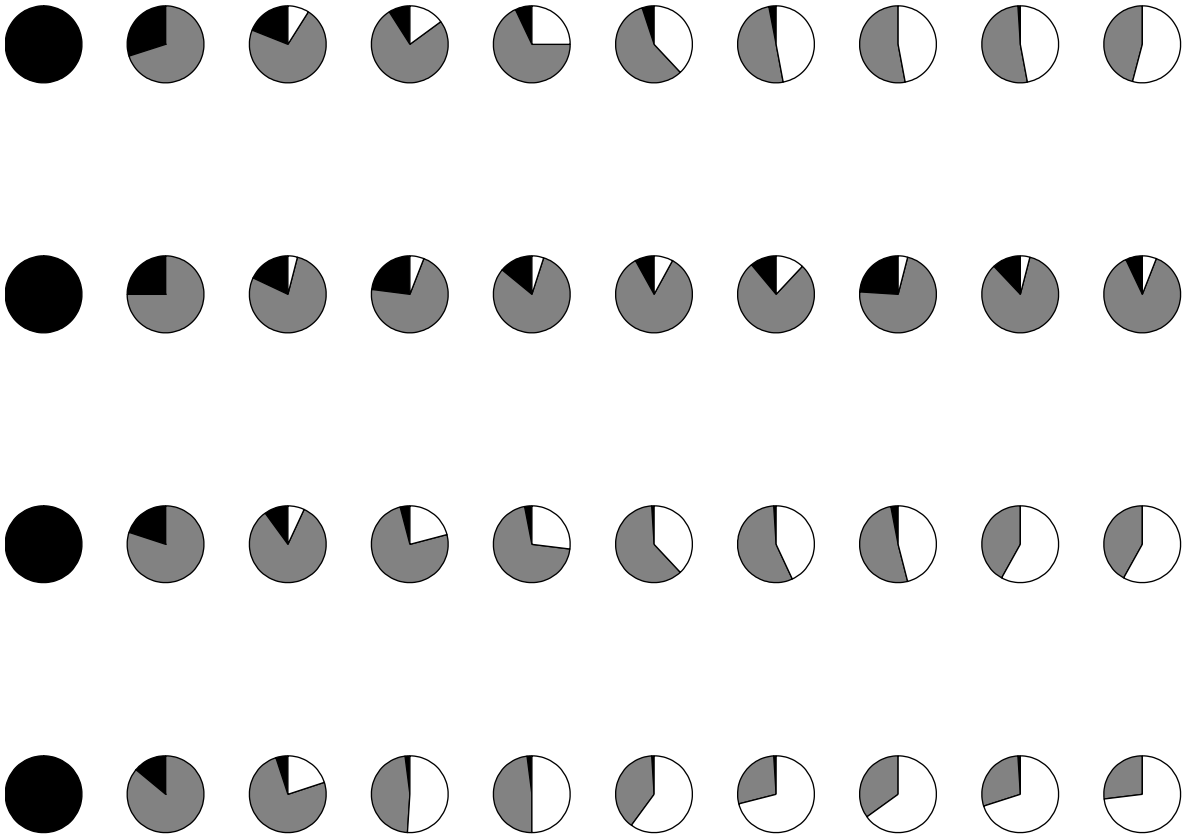


Figure 7.2: Performance of `sutton` local searches 1, 3, 8, and 11 with 1-4 memory states

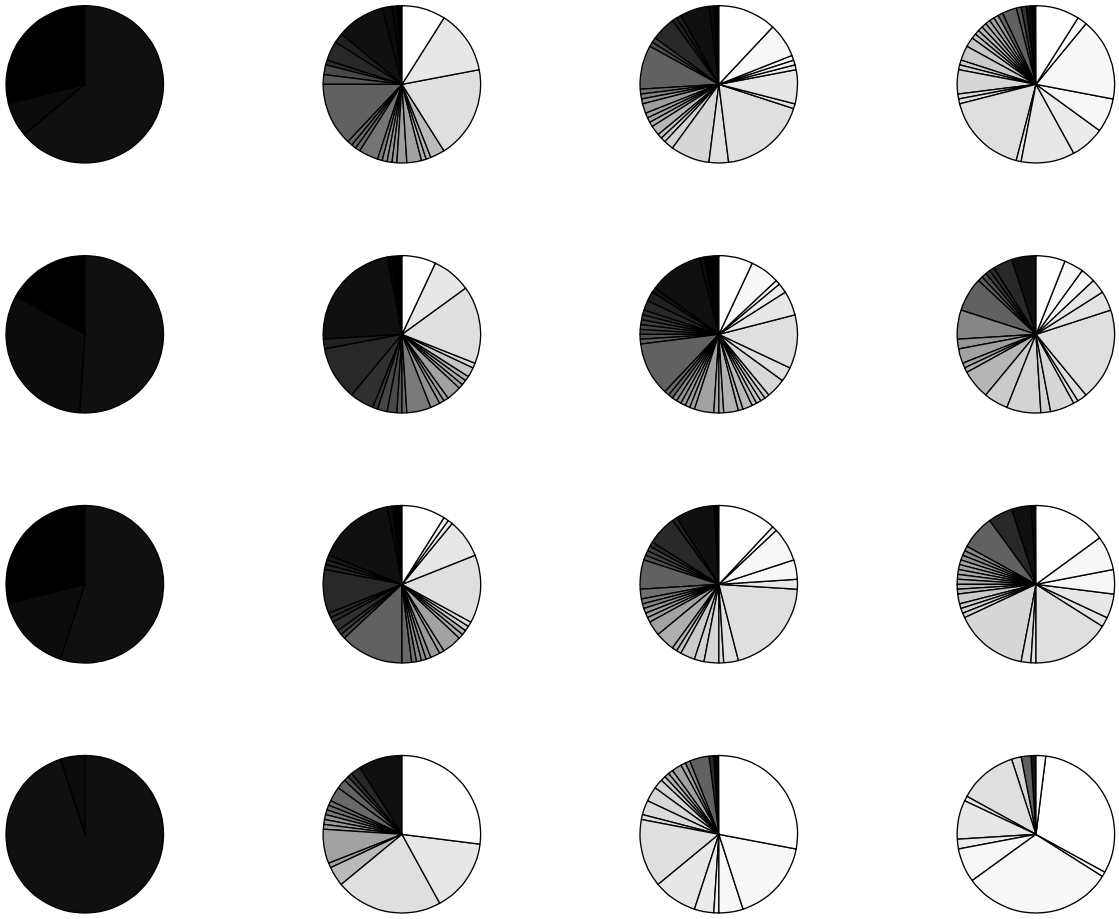


Figure 7.3: Performance of `aloha.30` local searches 1, 2, and 8, with 1-3 memory states

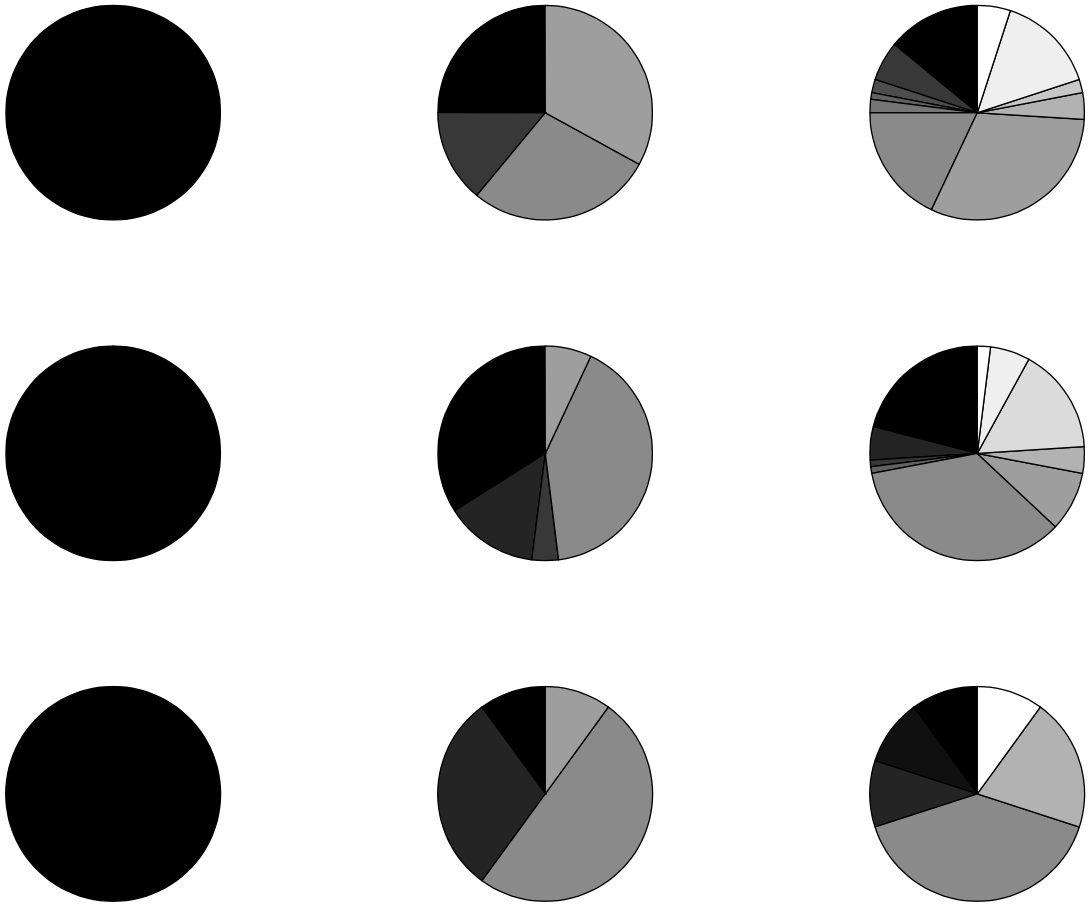


Figure 7.4: Performance of `1d` local searches 1, 9, 10, and 12, with 1-10 memory states

Figure 7.5: Performance of `sutton` local searches 1, 9, 10, and 12, with 1-4 memory states

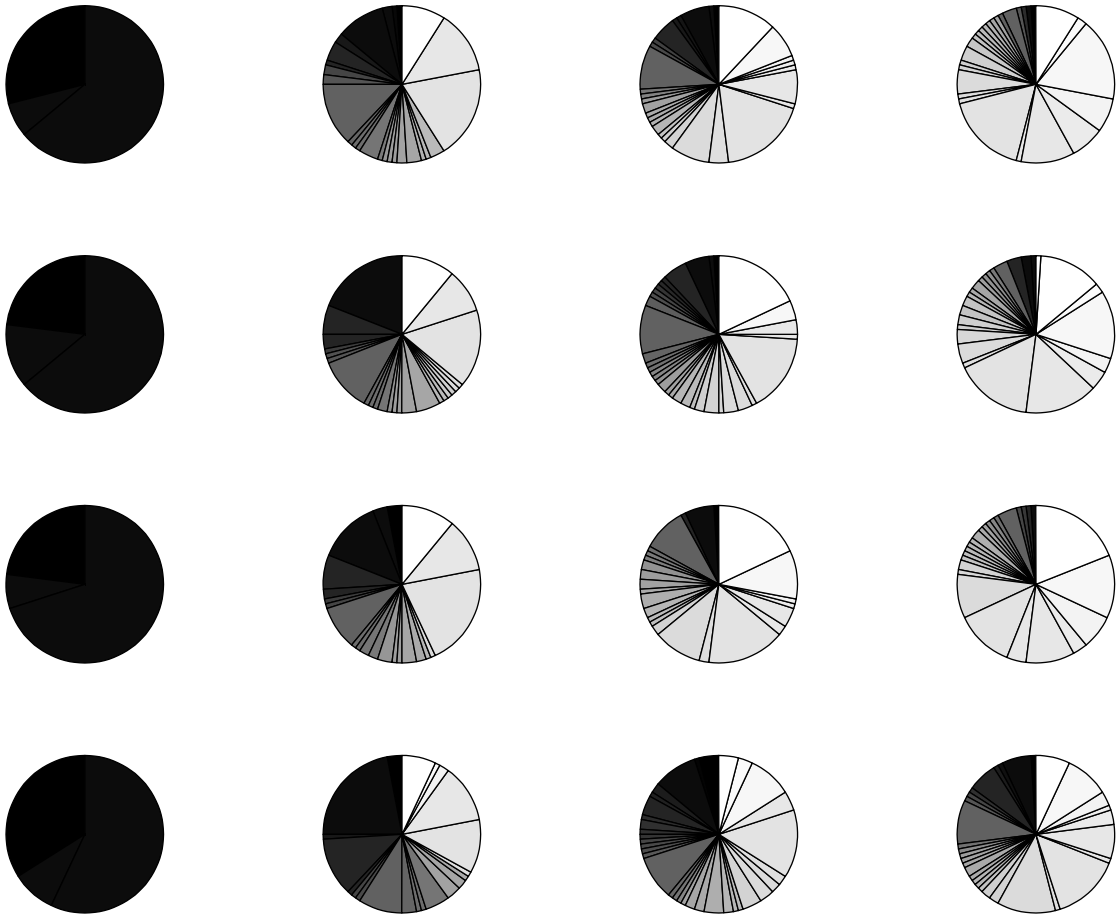


Figure 7.6: Time of `1d` local searches 1, 9, 10, and 12, with 1-10 memory states

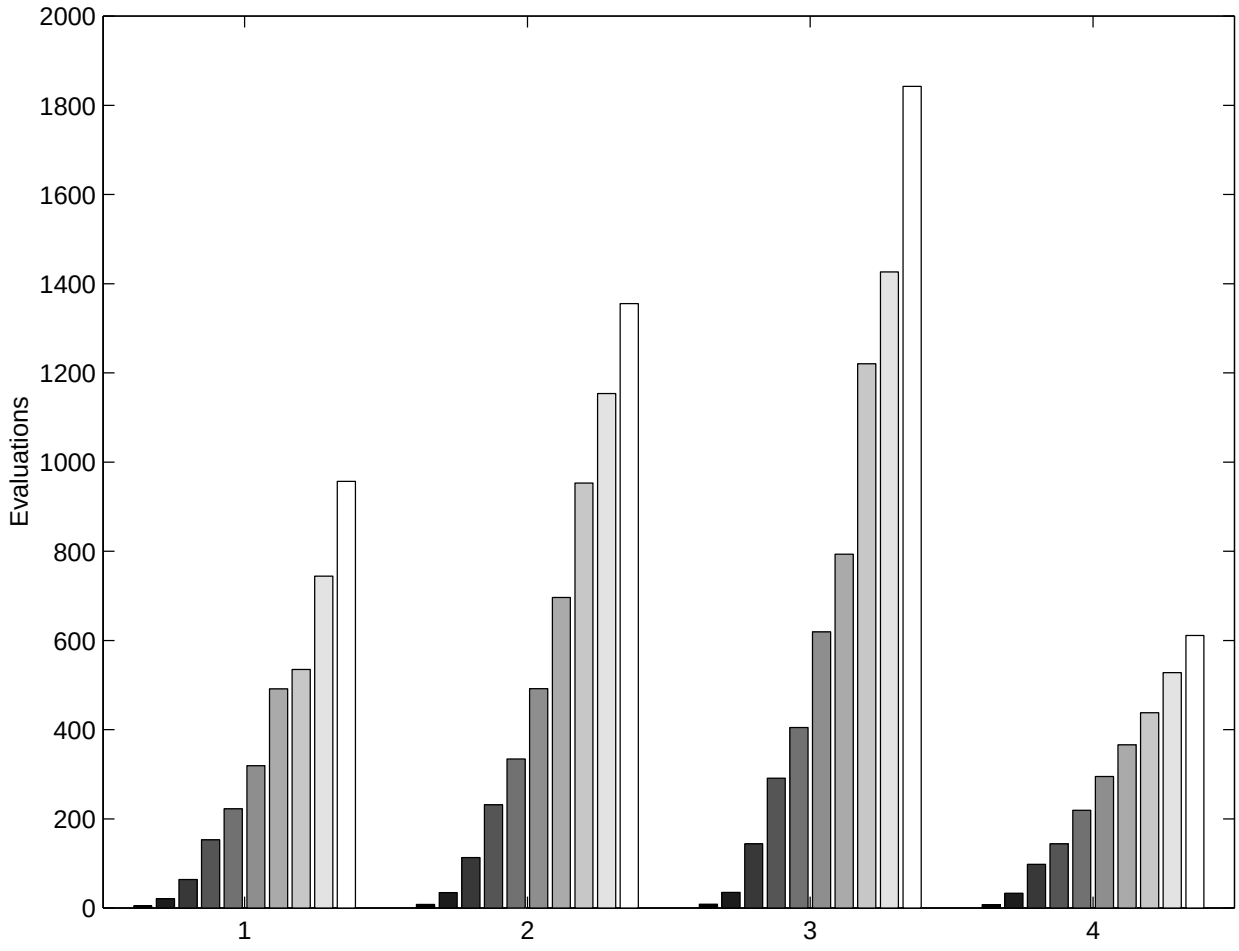


Figure 7.7: Performance of `sutton` local searches 1, 9, 10, and 12, with 1-4 memory states

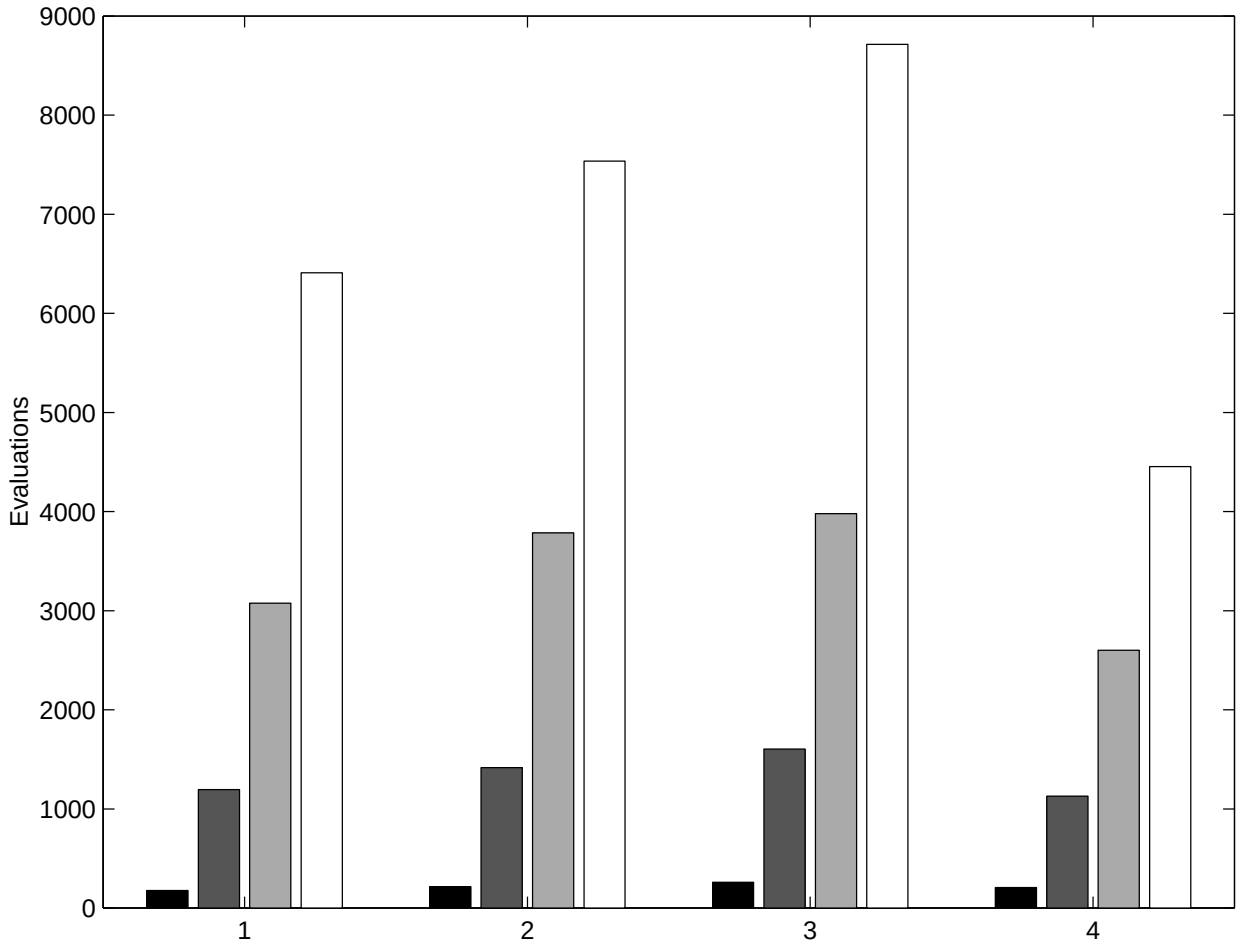
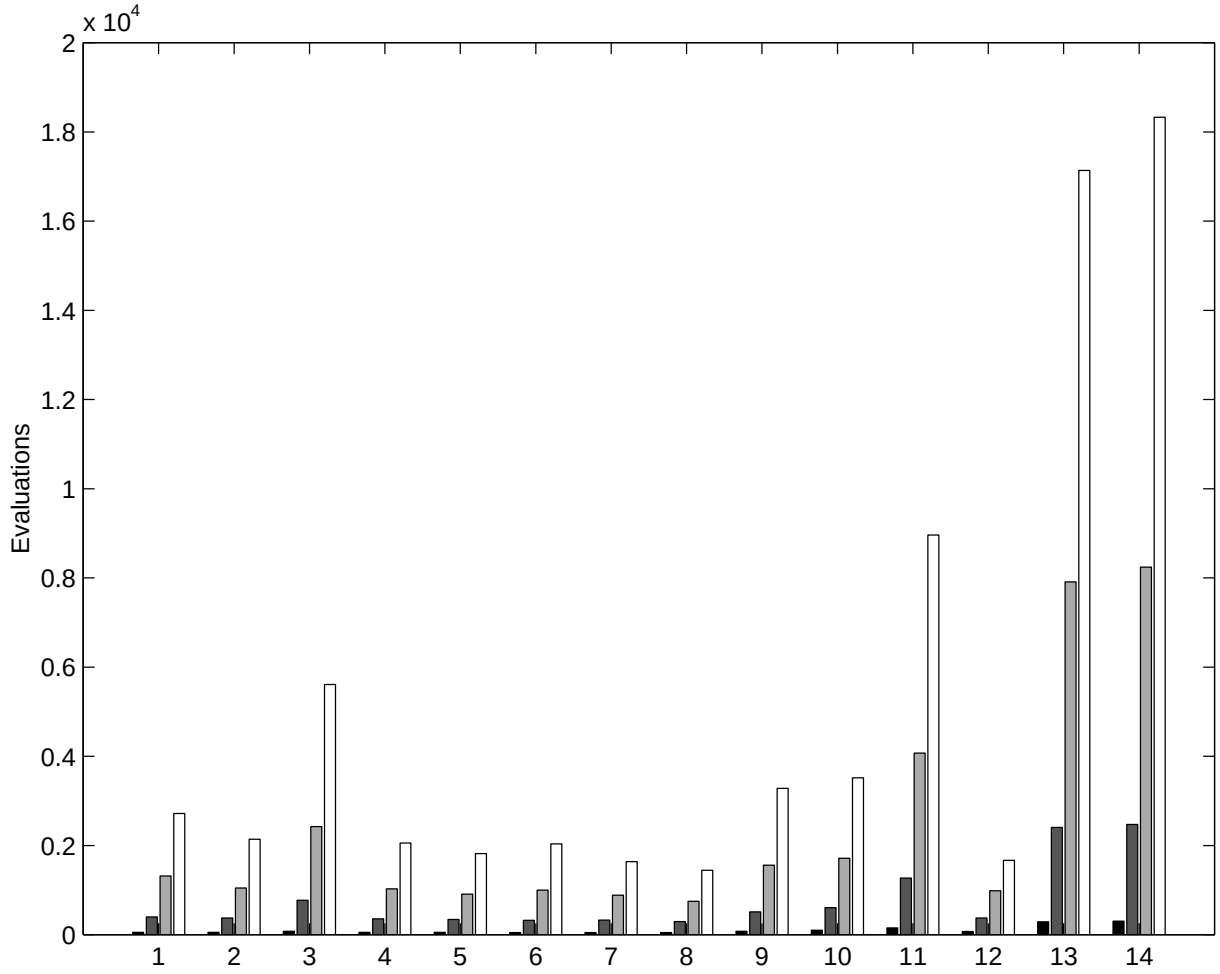


Figure 7.8: Time of `4x3.95` local searches 1, 9, 10, and 12, with 1-4 memory states



7.2.3 Best First and Variable-Depth Searches Are Slow

Not surprisingly, the searches that are forced to explore the entire neighborhood before choosing a neighbor are slower than those that have the option of cutting a neighborhood exploration short.

Timing data comparing all of the local search variants appears in Figures 7.2.3 and 7.2.3 for all variants of local search. Best first is variant 3, and the variable depth variants are 11, 13, and 14. The bar graphs are for `4x3.95`, `stand-tiger.95`, and `aloha10`. Notice that, for `aloha10`, variable depth search is actually *faster* than best first.

Figure 7.9: Time of `stand-tiger.95` local searches 1, 9, 10, and 12

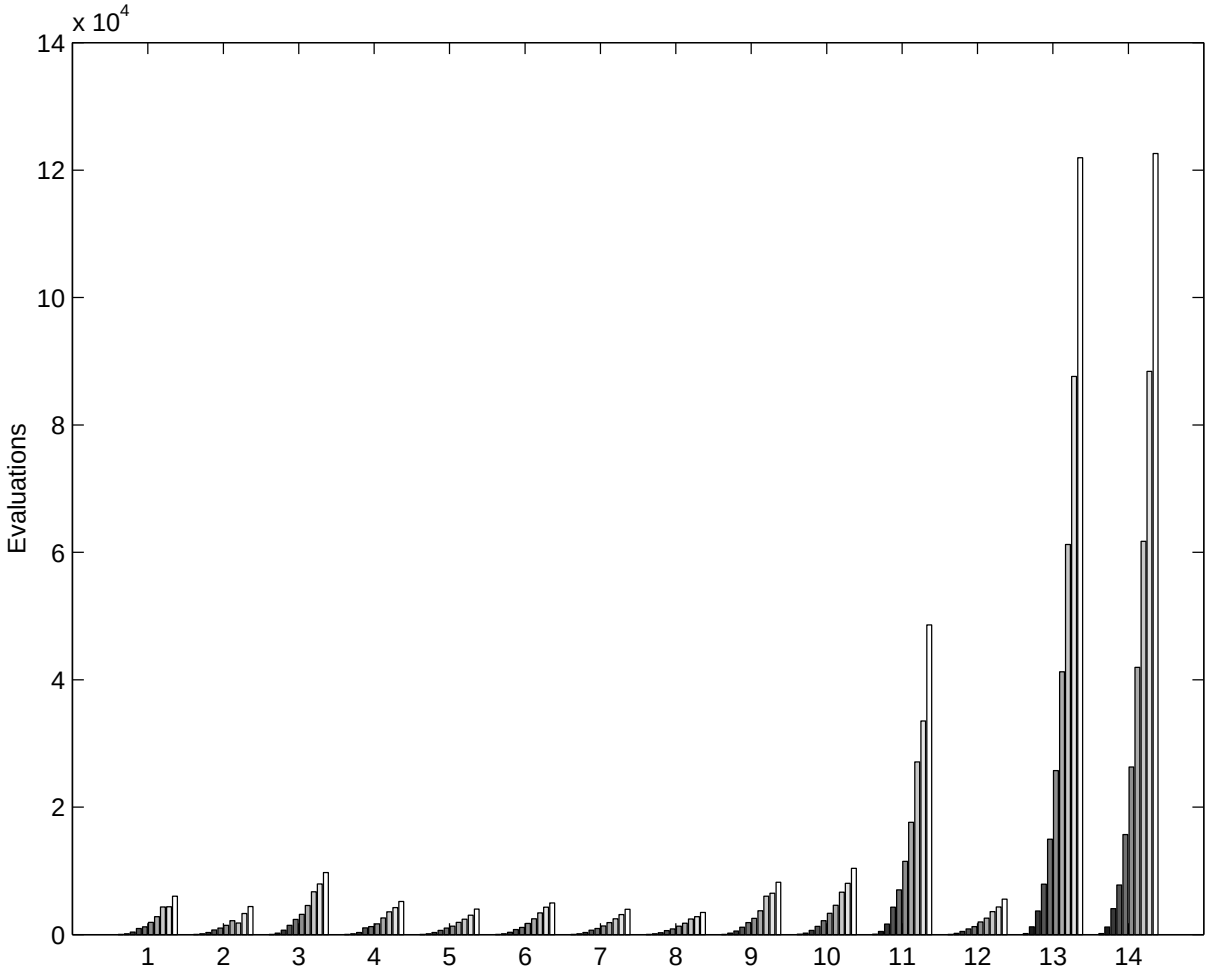


Figure 7.10: Time of `aloha10` local searches 1, 9, 10, and 12

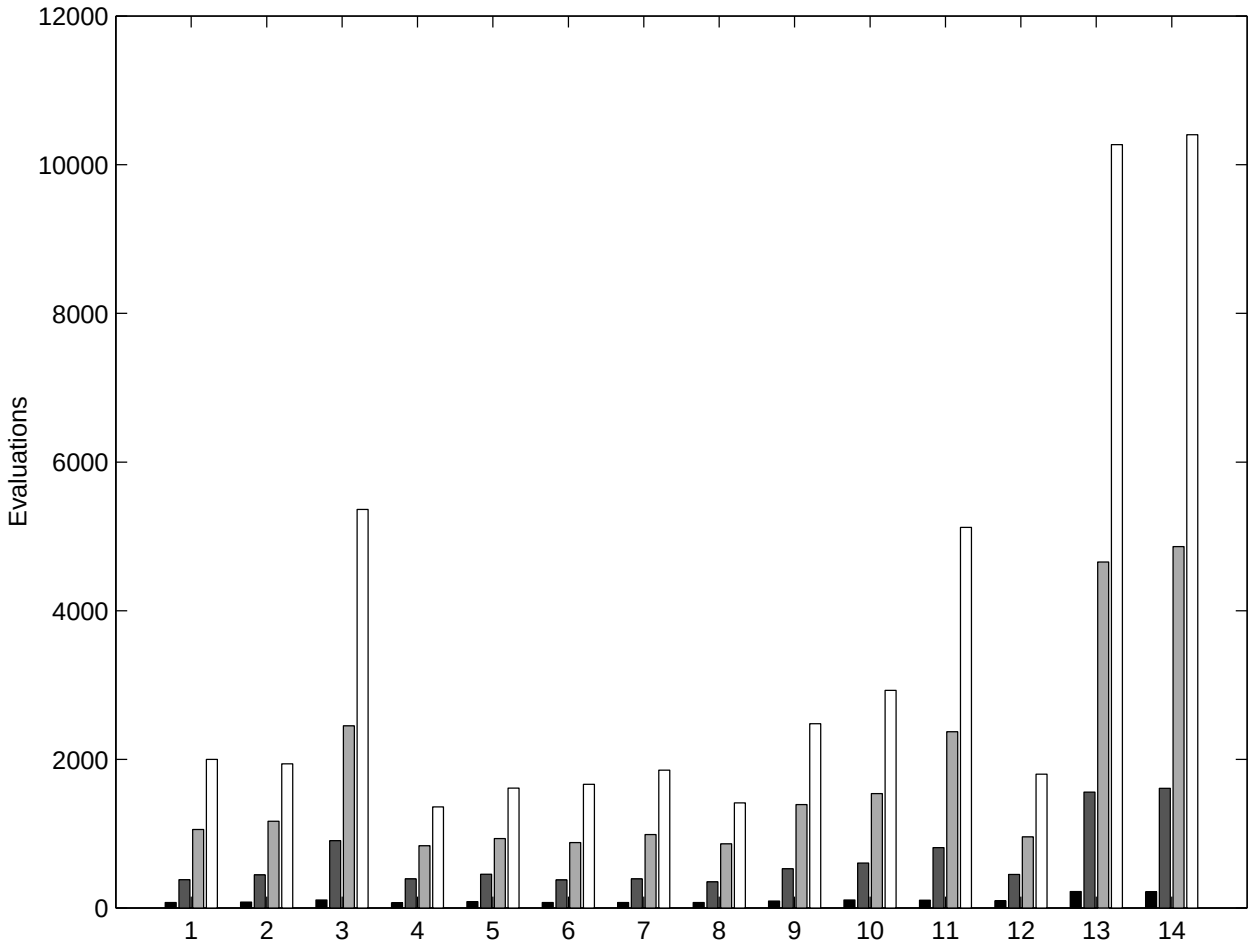
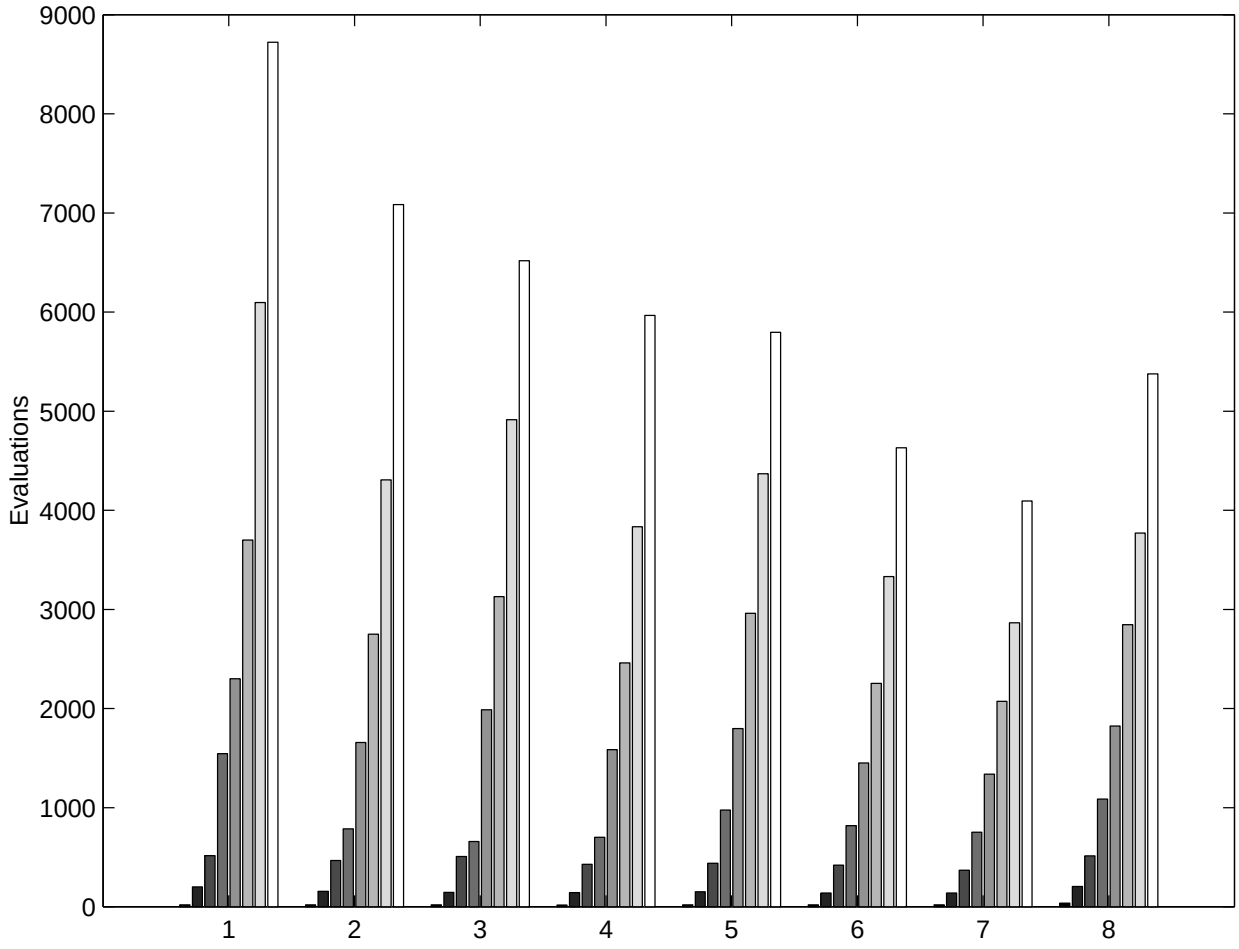


Figure 7.11: Time of `4x5x2.95` local searches 1, 2, 4, 5, 6, 7, 8, and 12



7.2.4 Acceleration Techniques Improve Speed, Damage Performance

The idea of local best was to speed up and/or improve policy quality by limiting the neighborhood search to the first table entry that showed some possibility of improvement. This technique succeeded in reducing run time, but at a cost of lowering average performance.

Timing and performance data for `4x5x2.95` are shown in Figures 7.2.4 and 7.2.4.

7.2.5 Variable Depth Search Wins on Performance

In all of the cases considered, the variable depth variants found more of the best-found policy values than the other variants. The local search variants shown are basic (1), depth 2 and 3 (9, 10), and variable depth (11, 13, and 14). Their performance is shown on `1d` and `sutton`

Figure 7.12: Performance of `4x5x2.95` local searches 1, 2, 4, 5, 6, 7, 8, and 12

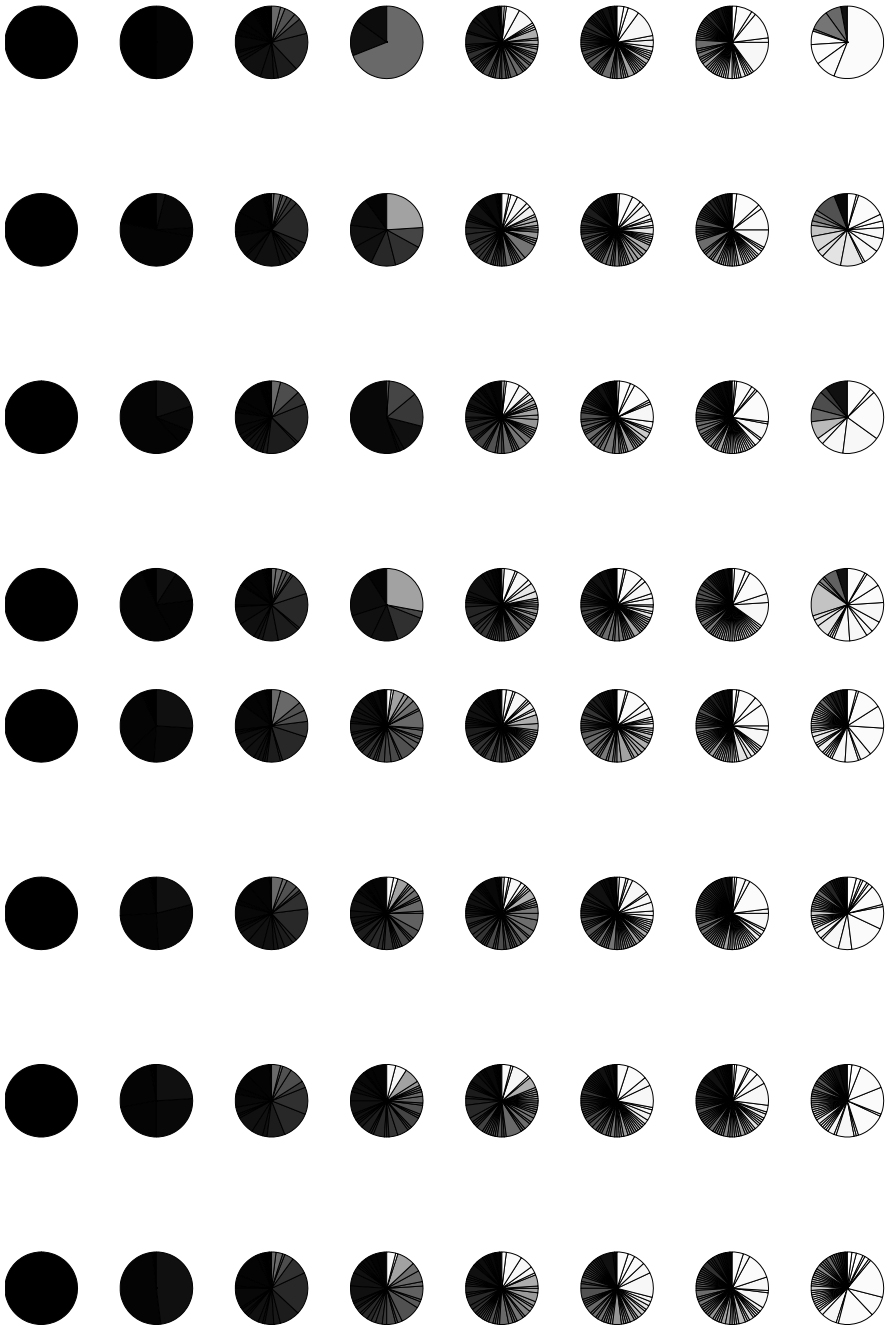


Figure 7.13: Performance of `1d` local searches 1, 9, 10, 11, 13, and 14

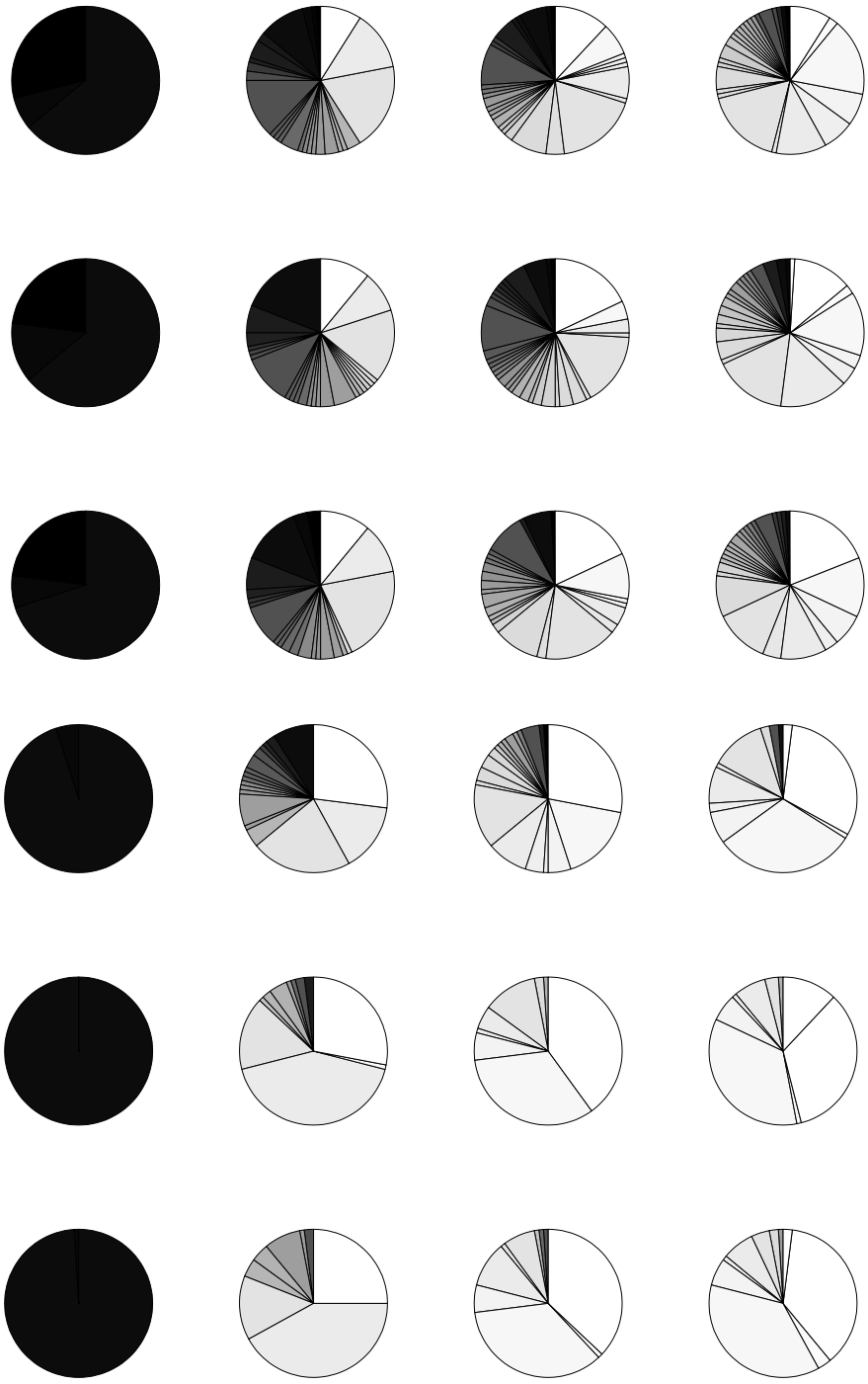
in Figures 7.2.5 and 7.2.5.

7.3 Recommendations For Partially Observable Markov Decision Process Algorithm Designers

In Lusena et al. (1999), Wells, Lusena, and Goldsmith (1999), we also explored simulated annealing and genetic algorithms for finding FFT policies for POMDPs. Neither of these approaches performed any better than the four variants of local search used in (Lusena et al., 1999).

Thus, we strongly encourage POMDP algorithm designers who are interested in finding a decent policy for their POMDP in reasonable time to run a local search variant with variable-depth lookahead, and to run it multiple times. Even with 100 iterations of the search, this is significantly faster than the current exact algorithms available. On the other hand, the

Figure 7.14: Performance of `sutton` local searches 1, 9, 10, 11, 13, and 14



policy found will have the limits on performance imposed by the limits set on memory size.

While astronomical improvements were implemented on the basic branch and bound heuristic, it is still not a viable option for large examples. Thus, an impatient POMDP algorithm designer will have to settle for taking the best policy found by this heuristic as optimal.

Chapter 8
Conclusions

Finding optimal or guaranteed near-optimal policies for POMDPs is intractible in the finite horizon case and uncomputable in the infinite horizon. Free finite table (FFT) policies often perform well, and it is not that complex to find good or optimal ones. Given a pre-set limit on the size of an FFT policy, one can realtively quickly find a reasonably good policy.

This dissertation makes contributions to areas of research on planning with POMDPs: complexity theoretic results and heuristic techniques. The most important contributions are probably the complexity of approximating the optimal history-dependent finite-horizon policy for a POMDP, and the idea of heuristic search over the space of FFTs.

References

Adel'son-Vel'skii, G., & Landis, Y. (1962). An algorithm for the organization of information. *Soviet Math Dokl.*, *3*, 1259–62.

Aoki, M. (1965). Optimal control of partially observed Markovian systems. *Journal of the Franklin Institute*, *280*, 367–368.

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

Bellman, R. (1957). *Dynamic programming.* Princeton University Press.

Boutilier, C., Dean, T. L., & Hanks, S. (1995). Planning under uncertainty: Structural assumptions and computational leverage. In *Proceedings of the second European workshop on planning.*

Boutilier, C., Dean, T. L., & Hanks, S. (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of AI Research*, *11*, 1–94.

Boutilier, C., Dearden, R., & Goldszmidt, M. (1995). Exploiting structure in policy construction. In *Extending Theories of Action: Formal Theory & Practical Applications: Papers from the 1995 AAAI Spring Symposium* (pp. 33–38). AAAI Press, Menlo Park, California.

Boutilier, C., & Poole, D. (1996). Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings 13th National Conference on Artificial Intelligence* (pp. 1168–1175). AAAI Press / MIT Press.

Brafman, R. I. (1997). A heuristic variable grid solution method for POMDPs. In *Proceedings of the 14th national conference on artificial intelligence and 9th innovative applications of artificial intelligence conference (AAAI-97/IAAI-97)* (pp. 727–733). Menlo Park: AAAI Press.

Cassandra, A. (1994). *Algorithms for partially observable Markov decision processes* (Tech. Rep. No. CS-94-14). Providence, Rhode Island: Brown University.

Cassandra, A. (1997–99). *Tony's POMDP page.* (`http://www.cs.brown.edu/research/ai/pomdp/index.html`)

Cassandra, A. (1998a). *Exact and approximate algorithms for partially observable Markov decision processes.* Unpublished doctoral dissertation, Brown University.

Cassandra, A. (1998b). *A survey of POMDP applications* (Tech. Rep. No. MCC-INSL-111-98). Austin, Texas: Microelectronics and Computer Technology Corporation (MCC). (Presented at the 1998 AAAI Fall Symposium: Planning with partially observable Markov decision processes.)

Cassandra, A., Kaelbling, L., & Littman, M. L. (1994). Acting optimally in partially observable stochastic domains. In *Proceedings of AAAI-94.*

Cassandra, A., Kaelbling, L., & Littman, M. L. (1995). *Efficient dynamic-programming updates in partially observable Markov decision processes* (Tech. Rep. No. CS-95-19). Providence, Rhode Island: Brown University.

Cassandra, A., Littman, M. L., & Zhang, N. (1997). Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes. In D. Geiger & P. P. Shenoy (Eds.), *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence (UAI-97)* (pp. 54–61). Morgan Kaufmann Publishers.

Cassandra, A. R., Kaelbling, L. P., & Kurien, J. A. (1996). Acting under uncertainty: Discrete Bayesian models for mobile-robot navigation. In *Proceedings of ieee/rsj international conference on intelligent robots and systems.*

Cheng, H. (1988). *Algorithms of partially observed Markov decision processes.* Unpublished doctoral dissertation, University of British Columbia.

Drake, A. (1962). *Observation of a Markov process through a noisy channel.* Unpublished doctoral dissertation, Dept. of Electrical Engineering, Massachussets Institute of Technology.

Hansen, E. A. (1997). An improved policy iteration algorithm for partially observable MDPs.

Hansen, E. A. (1998a). *Finite-memory control of partially observable systems.* Unpublished doctoral dissertation, Dept. of Computer Science, University of Massachussets at Amherst.

Hansen, E. A. (1998b). Solving POMDPs by searching in policy space. In *Proceedings of the Conference Uncertainty in Artificial Intelligence* (pp. 211–219).

Hauskrecht, M. (1997). *Planning and control in stochastic domains with imperfect information.* Unpublished doctoral dissertation, Massachusetts Institute of Technology.

Hauskrecht, M. (2000). Value-function approximations for partially observable Markov decision processes. *Journal of AI Research*, *13*, 33-94.

Hochbaum, D. (1997). *Approximation algorithms for NP-hard problems.* PWS Publishing Company.

Hopcroft, J., & Ullman, J. (1980). *Introduction to automata theory, languages, and computation.* N. Reading, MA: Addison-Wesley.

Howard, R. (1960). *Dynamic programming and markov processes.* MIT Press.

Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, *101* (1–2), 99–134.

Kim, K.-E., Dean, T. L., & Meuleau, N. (2000). Approximate solutions to factored Markov decision processes via greedy search in the space of finite controllers. In *Proceedings of the 5th international conference on artificial intelligence planning and scheduling* (pp. 323–330). Breckenridge, CO.

Lark III, J. (1990). *Applications of best-first heuristic search to finite-horizon partially observed Markov decision processes.* Unpublished doctoral dissertation, University of Virginia.

Li, T. (1999, April). *Implementing Hansen's algorithms for finite-memory control of partially observable systems.* (Master's Project)

Littman, M. L. (1994a). Memoryless policies: Theoretical limitations and practical results. In D. Cliff, P. Husbands, J.-A. Meyer, & S. W. Wilson (Eds.), *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior.* MIT Press.

Littman, M. L. (1994b). *The witness algorithm for solving partially observable Markov decision processes* (Tech. Rep. No. CS-94-40). Providence, Rhode Island: Brown University.

Littman, M. L. (1996). *Algorithms for sequential decision making.* Unpublished doctoral dissertation, Brown University.

Lovejoy, W. S. (1991a). Computationally feasible bounds for partially observed Markov decision process. *Operations Research*, *39*(1), 192–175.

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

Lusena, C., Goldsmith, J., & Mundhenk, M. (2001). Nonapproximability results for Markov decision processes. *Journal of Artificial Intelligence Research*, *14*, 83–103.

Lusena, C., Li, T., Sittinger, S., Wells, C., & Goldsmith, J. (1999). My brain is full: When more memory helps. In *Proceedings of the fifteenth Conference on Uncertainty in Artificial Intelligence* (pp. 374–381).

Madani, O., Hanks, S., & Condon, A. (1999). On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In *Proceedings of the sixteenth National Conference on Artificial Intelligence* (pp. 541–548).

Melekopoglou, M., & Condon, A. (1990). *On the complexity of the policy iteration algorithm for stochastic games* (Computer Science No. CS-TR-90-941). University of Wisconsin (Madison). (To Appear in the ORSA Journal on Computing)

Meuleau, N., Kim, K.-E., Kaelbling, L. P., & Cassandra, A. R. (1999). Solving POMDPs by searching the space of finite policies. In *Proceedings of the fifteenth Conference on Uncertainty in Artificial Intelligence* (pp. 417–426). Stockholm, Sweden.

Meuleau, N., Peshkin, L., Kim, K.-E., & Kaelbling, L. P. (1999). Learning finite-state controllers for partially observable environments. In *Proceedings of the fifteenth Conference on Uncertainty in Artificial Intelligence* (pp. 427–436). Stockholm, Sweden.

Monahan, G. (1982). A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Manag. Sci.*, *28*, 1–16.

Mundhenk, M. (2000). *The complexity of planning with partially-observable Markov decision processes* (Tech. Rep. No. TR2000-376). Dartmouth College.

Mundhenk, M., Goldsmith, J., Lusena, C., & Allender, E. (2000). Complexity results for finite-horizon Markov decision process problems. *Journal of the ACM*, *47*(4), 681–720.

Papadimitriou, C. H. (1994). *Computational complexity.* Addison-Wesley.

Papadimitriou, C. H., & Tsitsiklis, J. N. (1986). Intractable problems in control theory. *SIAM Journal of Control and Optimization*, *24*(4), 639–654.

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

Peshkin, L., Meuleau, N., & Kaelbling, L. P. (1999). Learning policies with external memory. In *Proceedings of the sixteenth International Conference on Machine Learning* (pp. 307–314). Morgan Kaufmann, San Francisco, CA.

Platzman, L. K. (1977). *Finite-memory estimation and control of finite probabilistic systems.* Unpublished doctoral dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Puterman, M. L. (1994). *Markov decision processes.* John Wiley & Sons, New York.

Puterman, M. L., & Shin, M. (1978). Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, *24*, 1127–1137.

Shamir, A. (1992). IP = PSPACE. *Journal of the ACM*, *39*(4), 869–877.

Smallwood, R. D., & Sondik, E. J. (1973). The optimal control of partially observed Markov processes over the finite horizon. *Operations Research*, *21*, 1071–1088.

Sondik, E. J. (1971). *The optimal control of partially observable Markov processes.* Unpublished doctoral dissertation, Stanford University.

Stockton, F. R. (1882). *The lady, or the tiger?* (available `ftp://ibiblio.org/pub/docs/books/gutenberg/etext96/ladyt10.txt`)

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction.* The MIT Press.

Thomas, L. C., Hartley, R., & Lavercombe, A. C. (1983). Computational comparison of value iteration algorithms for discounted Markov decision processes. *Operation Research Letters*, *2*, 72–76.

Thrun, S., Bennewitz, M., Burgard, W., Cremers, A. B., Dellaert, F., Fox, D., Hähnel, D., Rosenberg, C., Roy, N., Schulte, J., & Schulz, D. (1999). MINERVA: A tour-guide robot that learns. In W. Burgard, T. Christaller, & A. B. Cremers (Eds.), *Proceedings*

of the 23rd annual german conference on advances in artificial intelligence (KI-99) (Vol. 1701, pp. 14–26). Berlin: Springer.

Tseng, P. (1990). Solving h-horizon, stationary Markov decision problems in time proportional to $\log h$. *Operations Research Letters*, *9*(5), 287–297.

Wells, C., Lusena, C., & Goldsmith, J. (1999). *Genetic algorithms for approximating solutions to POMDPs* (Computer Science No. TR 290-99). University of Kentucky.

White, C. C., III. (1991). Partially observed Markov decision processes: A survey. *Annals of Operations Research*, *32*, 215–230.

White, C. C., III, & Scherer, W. (1994). Finite memory suboptimal design for partially observed Markov decision processes. *Operations Research*, *42*(3), 439–455.

Zhang, N., & Lee, S. (1998). Planning with partially observable markov decision process: Advances in exact solution method. In *Proceedings of the fourteenth Conference on Uncertainties in Artificial Intelligence.*

Zhang, N., & Liu, W. (1997a). A model approximation scheme for planning in partially observable stochastic domains. *Journal of Artificial Intelligence Research*, *7*, 199–230.

Zhang, N., & Liu, W. (1997b). Region-based approximations for planning in stochastic domains. In D. Geiger & P. P. Shenoy (Eds.), *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence (UAI-97)* (pp. 472–480). San Francisco: Morgan Kaufmann Publishers.

Zhang, N., & Zhang, W. (2001). Speeding up the convergence of value iteration in partially observable Markov decision processes. *Journal of AI Research*, *14*, 29–51.

Zhou, R., & Hansen, E. A. (2001). An improved grid-based approximation algorithm for POMDPs. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01).* Seattle, Washington.

CURRICULUM VITAE

Christopher Lusena

EDUCATION

1995 B.Sc. Hon.	Mathematics/Computer Studies, Trent University

EXPERIENCE

Fall 2000 – Spring 2001	Teaching Assistant University of Kentucky
Summer 1999 – Summer 2000	Research Fellowship, Department of Computer Science
Summer 1997 – Spring 1999	Research Assistant for Dr. Goldsmith
Fall 1996 – Spring 1997	Teaching Assistant University of Kentucky
Fall 1995 – Summer 1996	Research Assistant for Dr. Goldsmith

ACADEMIC HONORS AND AWARDS

Dissertation Fellowship, 1999
Dean's Honour List, Trent University 1995
Otonabee Scholar, Otonabee College, Trent University 1995
Otonabee Scholar, Otonabee College, Trent University 1994

PAPERS

"Complexity Results for Markov Decision Process Problems,"
with M. Mundhenk, J. Goldsmith, E. Allender,
Journal of the ACM, volume 47, number 4, pages 681–720, 2000

"Nonapproximability Results for Markov Decision Processes,"
with J. Goldsmith and M. Mundhenk.
Journal of AI Research, volume 14, pages 83-103, 2001

"My Brain in Full: When More Memory Helps,"
with T. Li, S. Sittinger, C. Wells, J. Goldsmith
Proceedings of the Conference on Uncertainty in AI '99, pages 374–381.

"Easy Versions of the Knapsack Problem"
UK CS Dept Technical Report 294-99 1999.

"Genetic Algorithms for Approximating Solutions to POMDPs,"
with C. Wells, J. Goldsmith
UK CS Dept Technical Report 290-99 1999.

In Preparation

"Finding Finite Memory Policies for POMDPs"
with C. Wells, J. Goldsmith.

RESEARCH PROJECTS

Spring 1997 – Present	Approximation Techniques and Optimal Finite Memory Policies for Partially Observable Markov Decision Processes
Spring 1996 – Present	The Complexity of Markov Decision Processes

REFEREE FOR

Information Processing Letters, IEEE Conference on Computational Complexity 1997

TALKS GIVEN

Conferences and Workshops

"Free Finite Memory Policies for Partially Observable Markov Decision Processes"
Midwest Theory Day, December 1999

"My Brain is Full: When More Memory Helps,"
Conference on Uncertainty in AI, August 1999 (poster)

"Nonapproximability Results for Markov Decision Processes,"
AAAI Fall Symposium Series, POMDP Symposium, October 1998

"Easy Versions of the Knapsack Problem,"
Midwest Theory Seminar, April 1998

Meetings and Local Presentations

"TEX and LATEX: Basic Use,"
For the local Linux User Group (UKLUG), February 1999, December 2000
For the student chapter of the ACM, October 1997

TEACHING

Spring 2001, CS 216 Introduction to Software Engineering
Fall 2000, CS 515 Algorithm Design and Analysis
Fall 1996 and Winter 1997, independent sections of CS 101, Introduction to Computing

Born: November 8, 1970, Hamilton, Ontario, Canada (Canadian Citizen)