Abstract of "Exact and Approximate Algorithms for Partially Observable Markov Decision Processes" by Anthony Rocco Cassandra, Ph.D., Brown University, May 1998

Automated sequential decision making is crucial in many contexts. In the face of uncertainty, this task becomes even more important, though at the same time, computing optimal decision policies becomes more complex. The more sources of uncertainty there are, the harder the problem becomes to solve. In this work, we look at sequential decision making in environments where the actions have probabilistic outcomes and in which the system state is only partially observable. We focus on using a model called a partially observable Markov decision process (POMDP) and explore algorithms which address computing both optimal and approximate policies for use in controlling processes that are modeled using POMDPs.

Although solving for the optimal policy is PSPACE-complete (or worse), the study and improvements of exact algorithms lends insight into the optimal solution structure as well as providing a basis for approximate solutions. We present some improvements, analysis and empirical comparisons for some existing and some novel approaches for computing the optimal POMDP policy exactly.

Since it is also hard (NP-complete or worse) to derive close approximations to the optimal solution for POMDPs, we consider a number of approaches for deriving policies that yield sub-optimal control and empirically explore their performance on a range of problems. These approaches

borrow and extend ideas from a number of areas; from the more mathematically motivated techniques in reinforcement learning and control theory to entirely heuristic control rules.

EXACT AND APPROXIMATE ALGORITHMS FOR PARTIALLY OBSERVABLE MARKOV DECISION PROCESSES

BY

ANTHONY ROCCO CASSANDRA

A.S., SUFFOLK COUNTY COMMUNITY COLLEGE, 1990

B.S., STATE UNIVERSITY OF NEW YORK AT STONY BROOK, 1992

M.SC., BROWN UNIVERSITY, 1994

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN THE DEPARTMENT OF COMPUTER SCIENCE AT BROWN UNIVERSITY

PROVIDENCE, RHODE ISLAND

MAY 1998

This dissertation by Anthony Rocco Cassandra is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ___________ ______________________________

Leslie Pack Kaelbling, Director

Recommended to the Graduate Council

Date ___________ ______________________________

Thomas L. Dean, Reader

Date ___________ ______________________________

Chelsea C. White, III, Reader

Approved by the Graduate Council

Date ___________ ______________________________

Peder J. Estrup

Dean of the Graduate School and Research

Vita

Name	Anthony Rocco Cassandra
Born	February 8, 1964 in Huntington, New York
Education	*Brown University*, Providence, RI Ph.D. in Computer Science, May 1998.
	Brown University, Providence, RI M.Sc. in Computer Science, May 1994.
	State University of New York at Stony Brook, Stony Brook, NY B.S. in Computer Science/Applied Math and Statistics, Summa Cum Laude, May 1992.
	Suffolk County Community College, Selden, NY A.S. in Computer Science, August 1990.
Honors	Award for Academic Excellence in Computer Science (SUNY at Stony Brook, 1992). Award for Academic Excellence in Applied Math and Statistics (SUNY at Stony Brook, 1992). Certificate of Excellence in Academics (SUNY at Stony Brook, 1991).

Acknowledgments

So many have helped, yet an attempt to acknowledge everyone runs the risk of overlooking someone. The alternative is to avoid acknowledgments altogether, but with all the help I have had, this hardly seems fair. Thus, at the risk of forgetting some, let me proceed to acknowledge those I haven't overlooked.

One of the earliest and most profound influences on my academic career came from Tenny Spofford in high school. His enthusiasm for learning and solving problems combined with his honesty, showed that education was about curiosity and open-mindedness and not about the same old regimented model of a high school classroom.

Non-destructive testing became my first career and there were many great people I have worked with over these years. This showed me that even the worst of jobs can be made bearable when the people around you are enjoyable. In particular, I should thank John Scalice for being a good friend during those times.

As my computer science academic career began, at night in Suffolk County Community College, there were many excellent teachers I was fortunate to have had. In particular, Morris Strongson and David Stampf's encouragement helped to further motivate me into pursuing a Bachelor's degree. My transition to full time studies at SUNY at Stony Brook benefited from the encouragement and consideration that Professor Peter Henderson provided. From his demanding, yet exciting introductory computer science course, through my graduation he remained supportive and provided me with my first experience in a research project.

If I had to point to a single person that was the direct cause of me entering graduate school, it would be Professor Alan Tucker. It was a conversation with Professor Tucker which prompted me to apply to graduate schools, something I would not have ever considered otherwise. Helping with my application effort, in addition to Professors Henderson and Tucker, were Professors Mishra and Stark who were all kind enough to agree to write me letters of recommendation, despite the short notice.

I now come to the hardest part in my acknowledgments: my time at graduate school in Brown's computer science department. My first officemates, Mitch Cherniak and Hagit Shatkay, became and remained good friends thought my time at Brown. Aside from friendship, Hagit has been a valuable resource for helping to work out many technical problems over the years. Although working in a different area, Mitch's expertise has often helped a great deal as well.

As far as technical contributions, not to mention friendship, Michael Littman deserves to most credit for the final result of my work. While still floundering in my early grad school days, his research experience, personality and general interest in the area I was working on, made for a unique relationship, which provided me with the opportunity to collaborate with and learn from him. My words cannot do justice to how important this was to my academic career.

The other close collaboration I had a Brown was with Jim Kurien. Working with Jim was a great experience; one which I would look forward to doing again. Although predominantly a long-distance collaboration, working with Nevin Zhang was also a pleasurable and productive experience.

Although most all the professors at Brown had helped to make my stay there pleasurable and exciting, I would especially like to thank Philip Klein and John Hughes for the time and efforts they contributed to my questions over the years.

My committee members Tom Dean and Chelsea White deserve thanks for agreeing to be on my committee and for providing me with useful feedback concerning the work contained in this thesis. Tom was especially influential in helping me get started when I was first at Brown looking for a research topic.

Lacking eloquent writing skills, I know I will not be able to properly thank my advisor Leslie Kaelbling for all she has done to help me over the years. Although it is hard for me to generalize, she was the ideal advisor for me. Her knowledge, down-to-earth approach, experience and patience are all things I very much appreciated and needed.

Finally, this work would never have been possible without my family: My mother and sisters, Debbie and Kim, for their constant support and my uncle Tom for teaching me many valuable things about life. Lastly, the two people I owe more thanks to than all of the previous people combined: my wife Ann Marie and my daughter Marie.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In a system where a human is the decision maker, the decisions can be based upon a myriad of factors: knowledge of the immediate circumstances; specialized knowledge; previous experiences about the effects of various actions; rules of thumb; established protocols; etc. For many situations, this approach works well or at least well enough that no one sees a need to change the manner in which decisions are made. However, as systems become more complex they impose a greater burden on a decision maker, since the various components and their interactions become more difficult to reason about. This difficulty only increases in systems where there is a high degree of uncertainty. This dissertation concerns itself with making sequences of decisions in the face of significant uncertainty.

One aspect of automated decision making is to ease the burden on the human by developing tools that are better able to cope with complex, uncertain processes or that can provide better decisions than their human counterparts. The field of operations research (OR) is one of the disciplines that has addressed problems from this perspective. A task such as inventory control for a large business has many complex interactions between customer demand, supply availability and monetary/physical resource limitations, all of which could easily overwhelm even the most dedicated corporate manager. Operations research has provided useful and successful tools to businesses for making these kinds of decisions.

Another aspect of automated decision making is motivated by problems where decisions have to be made, but it is not feasible or desirable to have a human available to make them. A particular sub-discipline of the artificial intelligence (AI) community has focused on automated decision making from this perspective. An interplanetary rover on a distant planet may not have time to communicate with earth-based controllers when it is faced with a navigational decision. With more autonomy, the rover can accomplish more, which is advantageous both from the financial perspective and to the scientific goals of the mission.

Naturally, regardless of the motivation for seeking automated decision making systems, the same basic problem is being addressed. It is no surprise that one finds many connections between areas in OR and AI. One area of AI is concerned with the problem of making a sequence of decisions over time given a model of the system. This area is typically referred to as *planning* and roughly corresponds to the problem of *optimal control* in OR. The problems that this thesis addresses fall roughly into the overlap between planning and optimal control.

Specifically, we are focused on sequential decision tasks where there are two major forms of uncertainty: the results of our decisions may not always have the same effects; and our perceptions of the system being controlled are not always very accurate. We will focus on a specific mathematical model that allows both of these forms of uncertainty to be modeled using probabilities.

The work in the OR community has been focused on the underlying theory with little emphasis on algorithmic development. The AI community has done much to develop algorithms, but often addresses simplified versions of the problems using ad-hoc strategies to patch their algorithms to enable them to handle the more general cases; specifically, there is often a deterministic assumption about the domains and dealing with uncertainty is an afterthought. In this thesis, we take a more algorithmic view of the OR research, motivated by work which originated in the AI community, and develop algorithms which have a basis in theory and account for uncertainty in a natural, motivated and consistent manner.

Sequential Decision Making Example

We will introduce a simple, yet illustrative example of a sequential decision making task which will help motivate the model in Chapter 2. This example is based on baseball, but is simple enough that only a vague familiarity with the sport is required. Appendix A gives a brief explanation of baseball for readers that are entirely unfamiliar.

Example We consider a very small portion of the decision making tasks of a baseball manager. In this problem, the manager has to decide when to remove the current pitcher as the game progresses. On any given day, the pitcher may be a favorable choice against a given opposing team, or the pitcher may be a poor choice.
The dynamics of a pitcher's ability and psyche as well as the opposing team's abilities and psyches are highly complex, so the manager does not necessarily know for sure whether the particular game's pitching match-up is favorable or not. However, as the game progresses, events during the game will provide hints about that day's match-up. Specifically, the manager knows when a pitcher has done a good job against a particular batter (e.g., the batter strikes out) and when they have not (e.g., the batter hits a home run).

This is a sequential decision making process, because as each opponent comes to bat, the manager must decide whether to let his pitcher pitch, or to replace the pitcher with someone from his pitching reserves in the bull-pen. There are immediate rewards and costs for any individual outcome, but the long term effects are what is most important; e.g., it is more important to win the game than to strike out a particular batter. This example is deliberately over-simplified and does not account for dozens of other factors that normally go into a baseball manager's decision, but its simplicity will allow us to illustrate the basic concepts of the model and algorithms much more lucidly.

Applications

Naturally, uncertainty is not limited to the game of baseball and there are many other real and important problems [1] in which decisions have uncertain outcomes and uncertain perceptions of the current state of the system. Below is just a small sample of problems of the type addressed in this thesis.

[1] We do not mean to imply that baseball is not a real and important problem.

Machine Maintenance A milling machine or lathe producing aircraft parts or a machine for assembling integrated circuits both have a myriad of internal components, all of which affect the tolerances and general quality of the parts being produced. However, the state of the internal components in the machine is not directly observable. There may be some general predictability based upon the age of the components or the number of operating hours, but parts wear and fail in a very non-deterministic fashion. Replacing the worn components before they produce defective parts is economically desirable, but accessing the actual state of the components requires disassembling the machine and a loss of revenue while the machine does not produce parts.

The decision task here is to develop a maintenance schedule: when to manufacture parts, inspect and/or replace internal components. This is not simply a matter of establishing a schedule such as every Tuesday being internal component inspection day, since the quality of the parts being produced provides an indirect, probabilistic observation about the internal components. If a machine is producing predominantly defective components on Thursday, waiting until Tuesday could cost the company a significant amount of revenue. Similarly, if the machine is still producing perfect parts all day Monday, it will be desirable and it may be possible to continue manufacturing parts on Tuesday, thereby saving the inspection costs. There has been a great deal of work using the models addressed in this thesis to address just such a problem [106, 99, 105] and we present a specific example of this in Appendix H.3, which we use in evaluating the techniques developed in this thesis. This inspection, maintenance and repair problem has a broader application than simply toward manufacturing machines; developing policies for infrastructure systems must also deal with stochastic state transitions (e.g., structural deterioration) and partially observable components (e.g., surface coatings mask the crucial structural components) [41].

Medical Diagnosis Doctors are constantly faced with sequential decisions making tasks under uncertainty [46, 125]. They must prescribe medicines and recommend tests or treatments based upon the internal state of the patient. However, accessing the true internal state of the patient is either impossible or highly undesirable, resulting is significant cost and risk to the patient. Lab tests provide some indication of the patient's internal state, but these are subject to errors and incur some cost. Additionally, the treatments prescribed do not always succeed or have varying results for different patients. Thus, the state of the system being controlled is only partially observable, though symptoms, lab tests, and the decisions (drugs, operations, etc.) are subject to probabilistic effects. Thus, determining good policies for patient diagnosis in the face of these uncertainties is a challenging, real and important problem. In addition to these individual patient decision-making tasks, the models used in this thesis are applicable to the higher level problem of developing health care system policies [115].

Computer Networks Although high-speed computer networks are likely to make significant bandwidth improvements in the coming years, the improvements in data storage and computer capabilities will continue to make the communication channel the major bottleneck in future information processing tasks. Additionally, the amount of information that will be electronically available will continue its explosive growth. There are, and will continue to be, a host of important decision making tasks in these domains [111, 10]. From routing decisions to distributed database queries, the need to account for uncertainty will be the key to robust systems. Regardless of the capacity of the network, the hardware will be prone to failure (e.g., power outages) the network configuration and the availability of information will change over time. Although there are many hints about the current configuration of the network, accessing the complete state of the system is often unneeded and/or undesirable due to the enormous bandwidth required to query all components in the network. Given that packets can get lost or dropped, the effects of a routing request or query will not necessarily have deterministic effects. Thus, computer or data networks are only partially observable and the results of decisions are not deterministic. Good policies for dealing with these uncertainties will translate into more efficient and productive network applications. We will present a particularly simple network application in Appendix H.2.

Other Domains The examples above are but a small portion of the domains where the techniques of this thesis are applicable. Additional applications include: cost control in accounting [56]; corporate structure internal audit timing [50]; learning processes [57]; teaching strategies [114]; moving target search [101]; fishery policies [64]; electric distribution network troubleshooting [123]; questionnaire design [128]; behavioral Ecology [77]; and elevator control [32].

Thesis Outline and Summary

The remainder of this thesis is organized as follows:

Chapter 2 presents the basic model for sequential decision making that will be used throughout this thesis. It breaks down roughly into an initial section that presents a simpler model and a following section which generalizes this model to the problems this thesis addresses. There are no new contributions in this chapter, since its purpose is to provide the general background required for the remainder of the thesis.

Chapter 3 discusses exact algorithms for solving problems formulated with the model defined in Chapter 2. It first develops the necessary background and concepts required to understand the nature of the algorithms and common issues that arise in all of them. The chapter then discusses two new algorithmic developments, which were jointly developed with other researchers, that have better theoretical and empirical properties than the previously existing algorithms. Here we present the algorithms, discuss some implementation concerns and show that they do indeed produce the correct answers. We conclude this chapter with discussion of some of the previous algorithms, including an intriguing variation of one algorithm that has previously received little attention, but which has interesting potential to be effective in practice.

In Chapter 4 we begin by reviewing the existing computational complexity results for the class of problems considered in this thesis. Here we see that a few different algorithms and variations of the algorithms lie in the same general complexity class. We then proceed with a more detailed analysis of the previous chapter's algorithms to more clearly define their differences. This will show that one of the novel variations developed here actually has an asymptotic improvement to the other algorithms. This chapter then discusses some miscellaneous issues and some minor optimizations to the algorithms which are only partially explored and developed at this time. We conclude this chapter with a series of empirical evaluations of these algorithms to validate the analysis, connecting the best and worst case complexity to some problem instances. We will see that the practice and the theory coincide nicely for these algorithms.

Chapter 5 presents our first approach to developing approximate solutions for these problems. It uses the techniques from reinforcement learning, shows some new approaches and develops some novel variations on existing approaches. We then present empirical results showing the potential benefits of these reinforcement learning schemes and discuss other variations which may add to the effectiveness of these techniques.

Our next approach to developing approximate solutions is discussed in Chapter 6 and concerns itself with heuristic approaches. While there is some underlying theory upon which these are based, it is not nearly as solid as the reinforcement learning approaches previously presented. Despite their lack of theoretical guarantees, these heuristics have the advantage of being extremely simple and fast, making them applicable to much larger problems than any of the previous approaches. We then compare and demonstrate the effectiveness of these heuristic approaches on a range of problems including the real problem of autonomous robot navigation. Aside from the effectiveness of these heuristics, we see that we can apply the models considered in this thesis to some real applications.

We conclude the main part of the thesis with a chapter of conclusions, contributions and future research directions. Following this is an extensive set of appendices, where some additional ideas and related concepts are discussed. Among them is a novel extension to an interesting class of solutions to these problems, though at this time it is unclear how useful this extension will prove to be.

Chapter 2

The Model

In this section we develop the formal model and review some of the well known results pertaining to this model. We use the terms *process* and *system* interchangeably to refer to the particular problem domain that is represented by the model.

The formal model we use is the Markov decision process (MDP) and is treated much more thoroughly in many texts [102, 9]. The model itself is fairly simple and it is only when trying to use these models to determine optimal behavior that any complications arise. We will first introduce a simpler version of the model, then discuss some of the existing theory and results for this simpler version, and finally discuss how to extend the MDP to handle the more general class of problems which this thesis addresses. The remainder of this thesis will look at algorithms for this more general class of problems, although further generalizations are possible [127, 129].

2.1 Markov Decision Processes

We are concerned with sequential decision problems where there is a need to make many decisions in the lifetime of the system. We assume that there is either a discrete, finite or infinite, sequence of time points at which we get to make decisions. It is possible to consider continuous time processes [102, 9], but we will not discuss the issues that arise from this added complexity.

> **Example** Each time a batter comes to bat is a decision point. Note that the "time" points are based more upon logical organization than upon some fixed increment of a clock.

Regardless of the system we are trying to model, we assume that at any given decision point in time, t, it is in one of a set of states, $\mathcal{S}$. We will use the random variable S^t to represent the state of the system at decision time t.

> **Example** The state of the process is whether or not the current pitching match-up is favorable. In this case we would denote the state set as $\mathcal{S} = \{\text{good}, \text{bad}\}$.

Since at each time point there is a decision to be made, the whole problem is to decide the proper *action* to take at a given time point. In the literature, *control* and *decision* are alternative names for the action choice. We define $\mathcal{A}$ to be the set of actions we have to choose from. We will use the random variable A^t to represent the action chosen at time t.

> **Example** In our simple example, the only choice facing the manager is either to leave the pitcher in or replace the pitcher with someone from the bullpen. Here we would define this action set to be $\mathcal{A} = \{\text{pitch}, \text{bullpen}\}$.

We assume that the set of states and the set of actions are both finite. Note that these assumptions are not necessary for MDPs in general, but without them the theory and the algorithms become much more complex [12, 13, 9]. We will see later a specific instance of an MDP with a continuous state set, but this will be the only time we consider MDPs without finite sets.

The system evolves as follows: at each time point, the system is in a particular state, s, an action a is taken and there is a transition to another state s'. However, we require that the state depend *only* upon s and a. In addition, s and a only give probabilistic information about what the resulting state will be.

Example When a manager decides to replace the pitcher, there are no guarantees that the next pitcher will have a good or bad match-up with the opposing team's batters. Thus, whether or not the current pitcher is a good match-up, the new state of the system when the next pitcher enters the game is equally likely to be good or bad.

To formally describe this evolution of states over time, we define the state transition function, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \Pi(\mathcal{S})$, to map each state-action pair into a probability distribution over the state space. We will use the notation $\Pr(S^{t+1} = s' | S^t = s, A^t = a) = \tau(s, a, s')$ for the individual transition probabilities. The fact that the next state probabilities only depend upon the current state and action is the Markov property of the process.

$\tau(s, \text{pitch}, s')$		s'	
		good	bad
s	good	0.9	0.1
	bad	0.0	1.0

$\tau(s, \text{bullpen}, s')$		s'	
		good	bad
s	good	0.5	0.5
	bad	0.5	0.5

Table 2.1: Transition probabilities, $\tau(s, a, s')$, for simplified baseball example.

Example Table 2.1 gives the state transition probabilities for our simple example. The transition function models the fact that occasionally a good match-up turns into a bad one during the course of a game. For instance, pitchers can get tired or start to suffer from some physical problem and could be reluctant to inform the manager of their condition. In this example, after each batter, there is a 10% chance that a good match-up becomes a bad match-up due to such factors. A more realistic example might have these probabilities dependent upon the time, since a pitcher is much more likely to tire later in the game than earlier. However, our example keeps things simple for expositional purposes. Also notice that a bad match-up never becomes a good match-up during the course of the game. We make the assumption that when the manager goes to the bullpen, the pitching match-up for the new pitcher is equally likely to be a bad or a good match-up. This could be altered to represent the manager's prior beliefs about the pitchers in his bullpen.

Thus far, there is nothing in the model to indicate that any one action should be preferred to another. We introduce a reward function that will indicate the immediate value of performing an action in a given state and then making a transition to some other state. We define the function, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, to be a real valued function over state-action-state triplets. Although this gives immediate values for guiding the action choices, we will normally be concerned with more long range effects of the decisions. We will discuss how we define the tradeoff between immediate and future rewards in Section 2.1.1 below.

Example In reality, replacing a pitcher can have a certain cost involved with it, since the manager has only a limited supply of pitchers and must ensure none of them ever pitches too often. Our model is too simple to build in anything as complex as this, so we use a much simpler model for the immediate rewards which we will discuss on page 32 when we discuss extending this basic model.

We will be able to represent the immediate reward slightly more succinctly with the function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that depends only on the current state and action chosen. With the transition probabilities, we can simply compute the expected immediate reward for a given state-action pair using

$$r(s, a) = \sum_{s' \in \mathcal{S}} \tau(s, a, s')\mathcal{R}(s, a, s') \ . \tag{2.1}$$

To summarize, the full model presented is defined as $\Xi = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T})$, where $\mathcal{S}$ is the set states, $\mathcal{A}$ is the set of actions, $\mathcal{R}$ is the immediate reward function and $\mathcal{T}$ is the state transition function. This model is a Markov decision process (MDP), though we will refer to it as a completely observable MDP (COMDP) to distinguish it from the more general, partially observable model we discuss in Section 2.3.

2.1.1 Optimality Criteria

The immediate reward function, $\mathcal{R}$ or r, helps to guide the decisions, but if we were simply interested in the immediate effects then, given the model, the problem would have a trivial solution of always choosing the action with the highest $r(s, a)$. The problem is more complex due to the trade-off between immediate short term rewards with the rewards that occur in the future.

There are many ways we could make the tradeoff between immediate and future rewards, but the one we will use is *expected future discounted* *reward*

$$E\left[\sum_{t=0}^{T-1} \rho^t r(S^t, A^t)\right] \quad , 0 \leq \rho \leq 1 \quad , \tag{2.2}$$

where S^t and A^t are the random variables for the state and action chosen at decision point t. Other optimality criteria used for making this tradeoff are discussed elsewhere [49, 48, 100, 102, 42], though not all are directly applicable to the methods discussed in this thesis.

With this criterion, rewards received later in time will have less value than an equivalent reward received closer to the present. The aim in solving the MDP is to find a control policy which maximizes this quantity.

Equation 2.2 specifies an expectation over T decision steps. When we are interested in optimizing this quantity, then we are solving a *finite horizon* problem where the horizon length is T. This criterion is adequate if the number of decision steps is known in advance. However, often the horizon length is not known in advance, or the decision process never actually terminates. For the indefinite or *infinite horizon* problem, we simply optimize with respect to the infinite sum

$$E\left[\sum_{t=0}^{\infty} \rho^t r(S^t, A^t)\right] \quad , \tag{2.3}$$

where now we must impose the constraint $0 \leq \rho < 1$ to ensure that the expectation is bounded. There are other criterion that can be used, such as average reward, which is sometimes more natural for a given problem. These tend to add complications to the theory and os we do not explore these alternatives here.

Aside from the mathematical convenience of yielding a finite sum, the discount factor often has more natural interpretations. When the problem is an indefinite horizon problem, the discount factor can be viewed as the probability that a subsequent decision will be required, i.e., the process terminates after each decision with probability $1 - \rho$. For problems with a more monetary basis, economic discounting over future returns becomes quite natural.

2.1.2 Solving MDPs

The model defined in Section 2.1 can be used to automatically determine the best choice of action to take at each point in time. However, the vast majority of MDP research makes the assumption that the actual state of the system at any time, S^t, is known to the decision maker when a decision is to be made. For many systems, this assumption is valid or close enough to correct to allow the results and algorithms to be useful.

We will briefly present some of this theory and algorithms for the completely observable case (COMDPs), though we are ultimately interested in problems with partially observable system states (POMDPs).

2.2 Completely Observable MDPs

In this section we discuss some of the theory and algorithms for solving completely observable Markov decision processes (COMDPs). In these processes the decision maker has access to the current state of the system at each decision point. Many more extensive and mathematically rigorous treatments have been given. [7, 49, 12, 102, 9].

2.2.1 Policies

The entire problem to be tackled in solving an MDP is to find a good policy based upon the past history, $\mathcal{H}$, of the process. This history will include the starting state, each subsequent state and the action taken up until the current decision point, t. Thus, a policy should provide us with an action to take based upon the previous history: $\mathcal{H} \rightarrow \mathcal{A}$. Note that the decision maker has no access to future events and must restrict its basis for decision to past information.

For an finite horizon problem, the number of possible histories is $(|\mathcal{S}||\mathcal{A}|)^{t-1}$ and for the infinite horizon problem there are an infinite number of histories. However, it can be shown that when the processes state is fully observable, optimal performance can be achieved by using only the current state to decide what action to take [102]. A policy that uses only the current state is called a *Markov policy* and all COMDP policies we will consider will be Markov.

We define a *decision rule* as a complete mapping from the set of states to the set of actions, $d^t : \mathcal{S} \rightarrow \mathcal{A}$. A complete Markov *policy* for a finite horizon COMDP is a sequence of decision rules, $\pi = (d^0, d^1, \ldots, d^{T-1})$, where d^t is the decision rule for the t^{th} time step. In general, we will want to know the best or optimal policy for an MDP and the process of determining the optimal policy is typically referred to as *solving* the MDP.

When the state set is finite, a decision rule can be easily represented with a finite-length vector of size $|\mathcal{S}|$. Since we also assume that the action set is finite, there is a finite number of different decision rules, $|\mathcal{A}|^{|\mathcal{S}|}$, exponential in the size of $\mathcal{S}$. Correspondingly, there is a large, but finite, number of finite horizon policies.

A policy where a different decision rule is applied for each time step is called a *non-stationary* policy and is typically required for optimal behavior in a finite horizon COMDP. A non-stationary infinite horizon policy poses a number of difficulties: not only are there an infinite number of such policies, but it may be impossible to represent such a policy using finite resources.

Therefore, for the infinite horizon case, it will be convenient to use a *stationary* policy, $\pi = (d, d, d, \ldots)$, where the same decision rule is applied at each decision point. It turns out that for an infinite horizon COMDP there is always a *stationary Markov policy* that is optimal. This overcomes both problems, since there is a finite number of such policies and we can represent the policy with a finite vector of size $|\mathcal{S}|$. Although we have used π for both stationary and non-stationary policies, in subsequent formula the context should prevent any confusion.

Policies where each decision rule completely determines the action to take are called *deterministic policies. Randomized* or *probabilistic* policies use a chance mechanism to decide on the action to choose. We will only need to consider deterministic policies since for the MDP models we consider an optimal deterministic policy always exists.

2.2.2 Value Functions

In this section we will briefly review the optimality equations for COMDPs which are covered with significantly more depth in many texts [102, 9] and early research papers [7, 49, 12]. The results in this section will serve as the basis for the remainder of the discussion.

Although the policy is the item of interest in solving an MDP, most of what we will discuss concerns itself with the value of a policy or the *value function*. We note that, although the optimal value function for an MDP is unique, there can be more than one policy that leads to the optimal value function. In general, we will be satisfied with any policy whose value function is optimal.

Finite Horizon

For a non-stationary finite horizon policy, π, we can compute the expected rewards for starting in a state s and following that policy for T steps. We define $V^t(\pi, s)$ as the value of starting in state s and executing the policy π for $T-t$ time steps. We can compute this value with the recursive equation

$$V^t(\pi, s) = r(s, d^t(s)) + \rho \sum_{s' \in \mathcal{S}} \tau(s, d^t(s), s') V^{t+1}(\pi, s') \quad , \tag{2.4}$$

starting with $t = 0$, ending the recursion with $t = T - 1$ and letting $V^T(\pi, s) = 0$ for all states s.

Evaluating a policy using Equation 2.4 directly, working from time 0 to time $T - 1$, results in a very inefficient procedure, since there is much duplication of effort down in the recursion tree. The preferred method takes

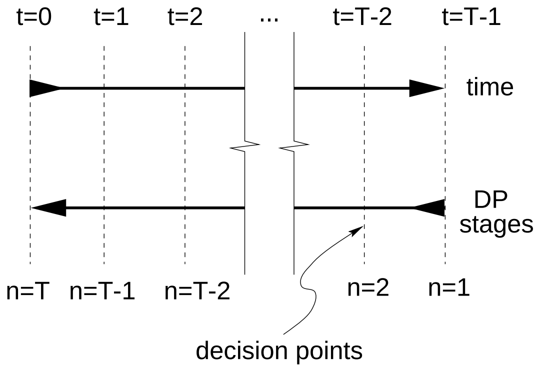


Figure 2.1: Relationship between time and DP indices.

advantage of the *principle of optimality* [7] and uses *dynamic programming* (DP) to compute a policy's value by working from time $t = T - 1$ down to decision time $t = 0$. When viewed "bottom up", Equation 2.4 is essentially the dynamic programming equation for determining the value of a policy though we prefer to use

$$V_n(\pi, s) = r(s, d_n(s)) + \rho \sum_{s' \in \mathcal{S}} \tau(s, d_n(s), s') V_{n-1}(\pi, s') \quad , \tag{2.5}$$

where we define $V_0(\pi, s) = 0$ for all states s.

Comparing Equations 2.4 and 2.5, notice the change from superscripted value functions and decision rules to subscripted value functions and decision rules. This is necessary, though often confusing, since dynamic programming works backwards in time. Figure 2.1 shows the relationship between time and the DP indices pictorially.

In Equation 2.4, $V^t(\pi, s)$ is the value of starting in state s when there are $T - t$ decisions to go, whereas $V_n(\pi, s)$ in Equation 2.5 is the value of starting in state s when there are n decisions to go. We will likewise use the interchanged indices in the decision rules so that $d^t = d_{T-t}$. Most of the remaining equations use the "number of steps to go" notation, but keep in mind that even though d_T will be the last decision rule computed for the finite horizon, in the execution of the policy it is the first one that would be used.

Infinite Horizon

There is a corresponding equation for the value of a stationary policy over the infinite horizon which is given, without a time index, by

$$V(\pi, s) = r(s, d(s)) + \rho \sum_{s' \in \mathcal{S}} \tau(s, d(s), s') V(\pi, s') \quad . \tag{2.6}$$

Note that the finiteness of the state set makes this a system of $|\mathcal{S}|$ equations with $|\mathcal{S}|$ unknowns. The nature of the model parameters and $0 \leq \rho < 1$ guarantees that this set of equations has a unique solution. A value function without a time index is assumed to be an infinite horizon value function.

Note that we could evaluate a stationary policy with Equation 2.5 over a finite horizon and, in fact, for a stationary policy π

$$\lim_{n \to \infty} ||V_n(\pi, s) - V(\pi, s)|| = 0 \quad , \tag{2.7}$$

where $|| \cdot ||$ is the supremum norm. In this case, Equation 2.5 is just a *successive approximation* scheme for solving the system of equations given in Equation 2.6. The proof of this uses the fact that the one-step DP operator is a contraction mapping when $0 \leq \rho < 1$, though we defer an explanation of this part of the theory to the more rigorous treatments [102, 9]. However, we will later use the fact that Equation 2.7 holds for both discrete and continuous space COMDPs.

2.2.3 Value Iteration

The dynamic programming approach does more than give us a way to evaluate a policy. By working from time $T-1$ to time 0, we can simultaneously compute the optimal policy and the optimal values. The intuition here is that with n steps to go, deciding on the optimal n-step policy is easy if we know what the optimal $n-1^{st}$-step policy, since we can simply consider the immediate next action for the current state and assume we know the optimal policy for the subsequent states. The additivity of the rewards in our optimality criteria and the Markov property makes this dynamic programming approach possible.

To compute the optimal value function for a finite horizon COMDP, we use the dynamic programming equation

$$V_n^*(s) = \max_{a \in \mathcal{A}} \left[r(s,a) + \rho \sum_{s' \in \mathcal{S}} \tau(s,a,s') V_{n-1}^*(s') \right] \quad , \tag{2.8}$$

where $V_n^*(s)$ represents the value of an optimal policy, π^*, when the starting state is s and there are n decision steps remaining. Note that the dynamic programming approach to finding the optimal value/policy in MDPs is referred to as *value iteration* (VI), or sometimes as *iteration in value space*.

It will be convenient to write the value function of Equation 2.8 in terms of other, related value functions

$$V_n^*(s) = \max_{a \in \mathcal{A}} V_n^{*,a}(s) \quad ,$$

where

$$V_n^{*,a}(s) = r(s,a) + \rho \sum_{s' \in \mathcal{S}} \tau(s,a,s') V_{n-1}^*(s') \quad . \tag{2.9}$$

```
valueIteration(Ξ, ρ, T)
    for each s ∈ S
        V_0(s) := 0
    end for each s
    for each n ∈ {1, 2, ..., T}
        V_n := oneStepDP(Ξ, ρ, V_{n-1})
    end for each n
    return V_T(·)
end valueIteration
```

Table 2.2: Routine for the value iteration algorithm.

The value $V_n^{*,a}$ has the interpretation: the value of performing action a with n steps remaining and performing optimally for the remaining $n-1$ steps. The functions will take on more significance in Chapter 5 where they will be referred to as *Q-functions* or *Q-factors*.

We can simultaneously compute the optimal policy $\pi^* = (d_T^*, d_{T-1}^*, \ldots, d_0^*)$ with

$$d_n^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} V_n^{*,a}(s) \quad .$$

Tables 2.2 and 2.3 shows the general structure of a routine for the value iteration algorithm for the COMDP model Ξ with discount ρ and a finite horizon T. Note that the value iteration routine only returns the last value function computed and that the policy, though computed, is never stored explicitly. The bookkeeping required to maintain the policy and/or the intermediate value functions is omitted to keep the exposition simple.

```
oneStepDP(Ξ, ρ, V)
    for each s ∈ S
        for each a ∈ A
            V^a(s) := r(s, a) + ρ Σ_{s'∈S} τ(s, a, s')V(s')
        end for each a
        V'(s) := max_{a∈A} V^a(s)
    end for each s
    return V'(·)
end oneStepDP
```

Table 2.3: Routine for one step of dynamic programming for a COMDP.

VI for the Infinite Horizon

There are many ways in which the infinite horizon value function can be computed, but one of these uses the same basic mechanisms of value iteration from the finite horizon problems. Recall that in section 2.2.2 we stated that an infinite horizon MDP is the limiting case of the finite horizon MDP.

Equation 2.6 defines the value of any policy, including the optimal policy, π^*. Since $V^*(s)$ is the value of following the optimal policy for an infinite number of steps, adding one additional step of rewards will not change its value and we have

$$V^*(s) = \max_{a \in \mathcal{A}} \left[r(s,a) + \rho \sum_{s' \in \mathcal{S}} \tau(s,a,s') V^*(s') \right] \quad . \tag{2.10}$$

From this we can compute an optimal infinite horizon policy, given the optimal value function with

$$d^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \left[r(s,a) + \rho \sum_{s' \in \mathcal{S}} \tau(s,a,s') V^*(s') \right] \quad ,$$

when the optimal stationary policy $\pi^* = (d^*, d^*, \ldots)$.

The main results of applying the value iteration algorithm to solve infinite horizon problems are that both the value function and the policy converge and that they converge to the optimal stationary policy and value function. The value function converges in the limit or can converge to be within some ϵ of the optimal values in a finite number of iterations for which loose upper bounds on the number of iterations exist. The policy always converges in a finite number of iterations and we can put a loose upper bound on the number of iterations required. The rates of convergence, stopping criteria, upper bounds and many other results relating to the convergence behavior of MDPs are interesting by themselves, though not discussed here [102]. We also note that these results apply to a much more general class of MDPs than the COMDPs thus far discussed.

2.2.4 Policy Iteration

The *policy iteration* (PI) method for solving infinite horizon COMDPs is based upon an iteration over policies and is sometimes referred to as *iteration in policy space*. We will discuss using the policy iteration idea in Chapter 5, but provide a simple overview of policy iteration for COMDPs here.

Although a policy's value function is a vector of values, it can be shown that there is guaranteed to be at least one policy π with the property: $\forall s, \forall \pi', V_\pi(s) \geq V_{\pi'}(s)$. Thus, a naive way to implement policy iteration is to iterate over the finite number of possible policies, use Equation 2.6 to compute their values and choose the policy with the highest value. However, this is terribly inefficient since there are an exponential number of policies.

The more efficient and practical approach, attributed to Bellman [7] and Howard [49], finds a sequence of policies of increasing quality and thus avoids the consideration of many suboptimal policies. This savings is purely empirical or average case, since it is possible to construct COMDPs where policy iteration would have to evaluate every possible policy. However, real problems tend not to have the structure required to force policy iteration into its worse case behavior.

The general structure of the policy iteration algorithm is shown in Table 2.4. Here the call to the function `evalPolicy`(π) is simply a routine that solves the systems of equations given by Equation 2.6. The routine `improvePolicy`(π) is given in Table 2.5 and amounts to using a greedy one-step look-ahead value calculation,

$$d'(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \left[r(s,a) + \rho \sum_{s' \in \mathcal{S}} \tau(s,a,s') V_\pi(s') \right] \quad ,$$

over all states where $\pi' = (d', d', \ldots)$. The routine of Table 2.5 uses the current value function and ensures that, when updating the policy, the policy only changes in states where an action is strictly better than the current action for that state.

The main result of applying this algorithm is that the sequence of policies generated by this algorithm is guaranteed to be monotonically increasing in value. Since there are a finite number of policies, this algorithm will converge on the optimal solution in a finite number of steps.

```
policyIteration(Ξ, ρ)
    d' := any decision rule
    do
        d := d'
        V := evalPolicy(Ξ, ρ, d)
        d' := improvePolicy(Ξ, ρ, V, d)
    until d = d'
    return d
end policyIteration
```

Table 2.4: Code fragment for the policy iteration algorithm.

```
policyImprovement(Ξ, ρ, V, d)
    for each s ∈ S
        for each a ∈ A
            V^a(s) := r(s, a) + ρ Σ_{s'∈S} τ(s, a, s')V_π(s')
        V(s) = max_{a∈A} V^a(s)
        if V(s) > V^{d(s)}(s)
            then d'(s) = argmax_{a∈A} V^a(s)
            else d'(s) = d(s)
        end for each a
    end for each s
    return d'
end policyImprovement
```

Table 2.5: Code fragment for the policy improvement routine.

2.3 Partially Observable MDPs

Although we can effectively solve COMDP problems, the solutions (policies) have limited use and generally cannot be applied when the system does not permit access to the state directly. A more general MDP model, a partially observable Markov decision process (POMDP), does not make the assumption that the states are directly observable. We will see that this added expressiveness comes at a significant cost in complexity.

> **Example** In our simple example, the manager does not know for sure whether or not they have a favorable or unfavorable match-up. It may have been an unfavorable match-up from the start due to factors of which the manager is unaware, or at some point the pitcher might become fatigued possibly causing a good match-up to become a bad match-up.

In a POMDP, we still assume that the system behaves in the same fashion as the MDP discussed previously: there are states, actions, rewards and state transitions based upon the current state-action pair. However, a set of observations, $\mathcal{Z}$, is added to the model so that after each state transition of the system, one of these observations is produced by the system and is accessible to the decision maker. The observation produced is correlated with the state transition, but does not generally allow us to completely determine the current state. We use the random variable O^t for the observation received at decision time t.

Example Although the manager does not have access to the underlying state, he can monitor the progress of the game and get some indication about whether or not the match-up is favorable. Again, for simplicity, suppose that each time a pitcher faces another batter, there are only two possible outcomes: a hit or an out. The game of baseball is not deterministic, so even the best pitchers give up hits. Likewise, even the worst pitchers can force some batters to make an out. However, if a pitcher is in a good match-up, then batters are less likely to get a hit than they would in a bad match-up. The manager can use these outcomes to gauge when they should remove the pitcher. Returning to the addition of the set of observations, in this example the observations are whether a batter gets a hit or makes an out. Thus, $\mathcal{Z} = \{\text{out}, \text{hit}\}$.

In addition to the observation set, we need to add an observation function, $\mathcal{O} : \mathcal{A} \times \mathcal{S} \to \Pi(\mathcal{Z})$, to the model. This function maps the action at time $t-1$ and the state at time t to a distribution over the observation set, which means that the observation is dependent upon the resulting state in the state transition[1]. We define $\Pr(O^t = z | S^t = s, A^{t-1} = a) = o(a, s, z)$ for the individual observation probabilities.

Example The observation probabilities for our example are given in Table 2.6 and show that these probabilities happen not to be dependent upon the action. They also show that a batter is more than twice as likely to get a hit in a bad match-up (0.350) than they are in a good match-up (0.150). This would roughly correspond to the opposing batters' batting averages for the two different states.

Finally, we note that the immediate reward function $r(s, a)$ is still applicable. However, recall that these were derived from a more expressive reward function where we used Equation 2.1 to define $r(\cdot, \cdot)$ as an expectation over the more general structure. This same technique can be applied in

[1]It is possible to make the observations dependent upon the initial state of the transition, or both the initial and final state, but these alternative formulations are expressively equivalent. [87]

$o(\text{pitch}, s', z)$		z	
		out	hit
s'	good	0.85	0.15
	bad	0.65	0.35

$o(\text{bullpen}, s', z)$		z	
		out	hit
s'	good	0.85	0.15
	bad	0.65	0.35

Table 2.6: Observation probabilities, $o(a, s', z)$, for simplified baseball example.

the POMDP model, though here we are allowed an even more general immediate reward structure. The most general form of a POMDP model's immediate reward, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{Z} \rightarrow \mathbb{R}$, makes the reward value dependent on every aspect of the state transition, including the observation received. We then define a simpler expected value version, similar to Equation 2.1, with

$$r(s, a) = \sum_{s' \in \mathcal{S}} \sum_{z \in \mathcal{Z}} \tau(s, a, s') o(a, s', z) \mathcal{R}(s, a, s', z) \ .$$

Also note that we will occasionally represent the immediate rewards with a column vector, $r(a)$, where the s^{th} component of the vector is $r(s, a)$.

$r(s,a)$		a	
		pitch	bullpen
s	good	0.065	-0.375
	bad	-0.925	-0.375

Table 2.7: Expected immediate rewards, $r(s,a)$, for simplified baseball example.

Example Because our example is so oversimplified, it is hard to derive a meaningful reward function. The various aspects of the game and the complicated interactions that a real baseball manager considers cannot be captured in such a simplistic model. Therefore, we will opt for a simple reward model where each hit and each out has some immediate reward. We define an out to have a value of 1.0 and a hit to have a value of -4.5. The motivation for this reward structure comes from an argument that uses the following facts: there are 27 outs in a game; on average, every two hits leads to a run; on average, if a pitcher gives up 3 or fewer runs in a game, then we consider it to be a good match-up. Therefore, over the course of a game, in a good match-up we would expect 6 hits. If we define a out to have value 1.0, then a game will result in a reward of $+27.0$ and if we want a bad game to have negative value, then more than 6 hits should cost us more than -27.0 in reward, thus 1 hit equals -4.5. Anything less than 6 hits and we would gain; anything more than 6 hits and we would lose.
In this example, we do not exploit the full generality available in the POMDP reward structure. In fact, the only dependency the rewards use is on the observations. Therefore, $\forall s, a, s'$ we have $\mathcal{R}(s,a,s',\text{out}) = 1.0$ and $\mathcal{R}(s,a,s',\text{hit}) = -4.5$. This translates into the expected immediate rewards shown in Table 2.7.

To summarize, the full POMDP model is formally defined by the 6-tuple $\Xi = (\mathcal{S}, \mathcal{A}, \mathcal{Z}, \mathcal{R}, \mathcal{T}, \mathcal{O})$ where: $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{T}$ are the same as their COMDP counterparts; $\mathcal{Z}$ and $\mathcal{O}$ are the observation set and related probabilities; and $\mathcal{R}$ generalizes the COMDP rewards to add a dependency on the observation. The system structure modeled by a POMDP is given in Figure 2.2.

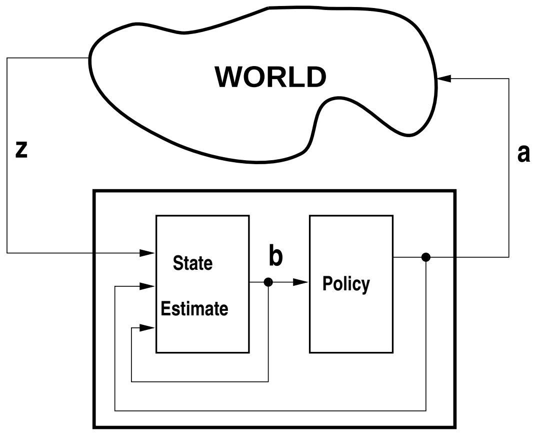


Figure 2.2: System structure for a system which can be represented using a POMDP.

2.3.1 Policies

As in any MDP, our goal is to find the optimal policy. From first principles, recall (from page 18) that, in general, a policy was defined over the entire history of the process: $\mathcal{H} \rightarrow \mathcal{A}$. The properties of a COMDP allowed us to simplify the problem to finding a simple mapping from states to actions which we called a Markov policy. A COMDP-type policy defined over $\mathcal{S}$ will be of little use when we do not have access to the system states.

Since it is generally infeasible to record the entire history of a process, and since a POMDP has many more possible histories, one approach is to look for a similar simplification used in the COMDP model. Since we only have access to the observation, we could consider defining a POMDP policy to be mapping from the last observation to an action, $\pi : \mathcal{Z} \rightarrow \mathcal{A}$. However, this type of policy can be very poor.

Example Supposed the manager in our baseball problem defined the policy: $d(\text{out}) = \text{pitch}$ and $d(\text{hit}) = \text{bullpen}$. Every hit will cause them to make a pitching change. If the pitcher got 15 outs in a row, then it is very likely to be a good match-up for the manager. Removing the pitcher after the first hit is not necessarily going to be desirable, since they are equally likely to end up with a bad match-up after the pitching change.

For a POMDP there may be no policy of this type which is optimal [51]. Some research exploring policies based upon the last observation only can be found in work by Littman [69].

A slightly better approach than the deterministic observation-based policy is to define the policy, $\pi : \mathcal{Z} \rightarrow \Pi(\mathcal{A})$, as a mapping from observations to a distribution over actions. This is commonly referred to as a *probabilistic* policy and is slightly more general since it includes the deterministic observation-based policies. Although there has been some research into these types of policies [51], these too can be arbitrarily poor.

It turns out that the optimal policy for a POMDP is not necessarily a Markov policy with respect to observations or any finite history of observations. In general, to behave optimally in a POMDP, the policy must be able to remember the entire history of the process. Since this history can be arbitrarily long, one might imagine that events far in the past might have minimal effects on our current decision. For this reason, researchers have explored policies based on a finite history of the process [133]. Although these methods can yield good solutions, they too can be arbitrarily poor. As a example of how a finite history can be poor, consider the simple case where knowledge of some sort of parity in the history will be required to behave optimally. No matter what finite history size is chosen, information will be lost and optimal behavior will be unachievable. The only way for a policy to specify truly optimal behavior is for it to remember the entire history.

2.3.2 Information States

As mentioned, optimal behavior in a POMDP is going to require access to the entire history of the process. It is possible to derive a summary statistic for the entire history of a process. We will refer to this statistic as an *information state* or *belief state*. Unlike the entire history, the information state size is of fixed dimension. An information state is a *sufficient statistic* for the history, which means that optimal behavior can be achieved using the information state in place of the history [120, 2, 117].

An information state, b, is simply a probability distribution over the set of states, $\Pi(\mathcal{S})$, with $b(s)$ being the probability of occupying state s. We define $\mathcal{B} = \Pi(\mathcal{S})$ to be the space of all probability distributions over $\mathcal{S}$. A single information state can capture the relevant aspects of the entire previous history of the process, and more importantly can be easily updated after each state transition to incorporate one additional step into the history.

We can derive the formula for updating an information state from first principles using basic rules from probability theory, Bayes rule and the independence assumptions inherent in the POMDP model. Given a information state vector b, we would like to compute the resulting information state, b_z^a, after a transition in the process, which, for the s^{th} component, is derived

$$\begin{aligned}
b_z^a(s') &= \Pr(s'|b,a,z) \\
&= \frac{\Pr(s',b,a,z)}{\Pr(b,a,z)} \\
&= \frac{\Pr(z|s',b,a)\Pr(s'|b,a)\Pr(b,a)}{\Pr(z|b,a)\Pr(b,a)} \\
&= \frac{\Pr(z|s',a)\Pr(s'|b,a)}{\sum_{s,s''}\Pr(z|b,a,s,s'')\Pr(s,s''|b,a)} \\
&= \frac{\Pr(z|s',a)\sum_s \Pr(s'|b,a,s)\Pr(s|b,a)}{\sum_{s,s''}\Pr(z|a,s'')\Pr(s''|b,a,s)\Pr(s|b,a)} \\
&= \frac{\Pr(z|s',a)\sum_s \Pr(s'|a,s)\Pr(s|b)}{\sum_{s,s''}\Pr(z|a,s'')\Pr(s''|a,s)\Pr(s|b)} \quad ,
\end{aligned}$$

where $\Pr(s|b) = b(s)$, $\Pr(s''|a,s) = \tau(s,a,s'')$ and $\Pr(z|s'',a) = o(a,s'',z)$ making the information state update equation

$$b_z^a(s') = \frac{o(a,s',z)\sum_s \tau(s,a,s')b(s)}{\sum_{s,s''} o(a,s'',z)\tau(s,a,s'')b(s)} \quad . \tag{2.11}$$

By basing the POMDP policies on the information state, we have regained the Markov property for our policy: the next information state depends only upon the previous information state and the immediate transition taken (i.e., the action and observation received). In fact, the information state process is itself a Markov process, which we will discuss in Section 2.3.3.

Information State Equations

It will be convenient to define some extra notation related to information states that allows concise formulation of the optimality equations and algorithms. We define the conditional probability of an observation as

$$\begin{aligned}\sigma(b,a,z) &= \Pr(z|b,a)\\ &= \sum_{s\in\mathcal{S}}\sum_{s'\in\mathcal{S}} b(s)\tau(s,a,s')o(a,s',z)\ ,\end{aligned} \tag{2.12}$$

where we are conditioning on the current information state and action choice. Note that this simplifies Equation 2.11 to

$$b_z^a(s) = \frac{o(a,s,z)\sum_{s'\in\mathcal{S}}\tau(s',a,s)b(s')}{\sigma(b,a,z)}\ . \tag{2.13}$$

Next, we define the state transition probabilities for information states. The state transition function defines the probability of a particular successor information state, given an initial information state and action. Since each observation can yield a different succeeding information state, the information state transitions can be specified

$$\psi(b,a,b') = \sum_{z\in\mathcal{Z}} \sigma(b,a,z)I(b',b_z^a)\ , \tag{2.14}$$

where

$$I(x,y) = \begin{cases} 1 & \text{if } x=y \\ 0 & \text{otherwise.}\end{cases} \tag{2.15}$$

In words, the probability of an information state is the sum of the probabilities of all the observations that would lead to this information state.

2.3.3 Value Functions

The most interesting result concerning the use of information states is that, having regained the Markov property, the POMDP can be reformulated as a continuous space COMDP [2, 1, 109]. The fact that Equations 2.8 and 2.10

still apply to the continuous space problem (as do the related equations) means that we can borrow many of the theoretical results and algorithmic ideas to apply to the POMDP problem.

To actually derive the dynamic programming equation from Equation 2.8, we need to describe the full transformation of the discrete space POMDP into the continuous space COMDP. The state space for this continuous space problem is the information space $\mathcal{B}$ and the action set is the original POMDP action set.

Given an information state, since the action set and observation set are finite, there are only a finite number of possible successor information states. The state transition function for information states is given by Equation 2.14. We define the set of possible successor states as

$$\mathcal{B}'(b,a) = \{b_z^a | z \in \mathcal{Z}\} \ .$$

The rewards for our information state COMDP need to be defined for each state-action pair, which in this case means for each information state and action. Since the POMDP rewards are based upon actual POMDP states, the reward for a information state is

$$\omega(b,a) = \sum_{s \in \mathcal{S}} b(s) r(s,a) \ , \tag{2.16}$$

which simply uses the information state in an expectation over all states.

We can now make the following substitutions in Equation 2.8: b for s; $\mathcal{B}'(b,a)$ for $\mathcal{S}'$; $\psi(b,a,b')$ for $\tau(s,a,s')$; and $\omega(b,a)$ for $r(s,a)$. This yields the dynamic programming equation for a POMDP,

$$V_n^*(b) = \max_{a \in \mathcal{A}} \left[\omega(b,a) + \rho \sum_{b' \in \mathcal{B}'(b,a)} \psi(b,a,b') V_{n-1}^*(b') \right] \quad . \tag{2.17}$$

From Equation 2.14, it follows that the sum over the possible successor information states can be replaced with a sum over all observations, so that

$$V_n^*(b) = \max_{a \in \mathcal{A}} \left[\omega(b,a) + \rho \sum_{z \in \mathcal{Z}} \sigma(b,a,z) V_{n-1}^*(b_z^a) \right] \tag{2.18}$$

is equivalent to Equation 2.17.

Although this is a continuous space COMDP and much of the existing work on continuous space problems would be applicable, there are certain properties of the converted POMDP problem that can be exploited, which can make it amenable to techniques not available for general continuous space COMDPs. One property is that, though the state space is continuous, the number of succeeding states is finite.

Here we rewrite Equation 2.17 in the more explicit, though more cluttered, manner which uses the POMDP model parameters directly:

$$\begin{aligned} V_n^*(b) = \max_{a \in \mathcal{A}} & \sum_{s \in \mathcal{S}} b(s) r(s,a) \\ & + \rho \sum_{s \in \mathcal{S}} \sum_{s' \in \mathcal{S}} \sum_{z \in \mathcal{Z}} b(s) \tau(s,a,s') o(a,s',z) V_{n-1}^*(b_z^a) \quad . \end{aligned} \tag{2.19}$$

We re-emphasize that Equations 2.17, 2.18 and 2.19 are all equivalent and the infinite horizon optimality equations for a POMDP can be adapted in the same manner though it would use Equation 2.10 as the basis for the conversion.

For many of the algorithms discussed in Chapter 3, it will help to break down the optimal finite horizon POMDP value function of Equation 2.18 into a series of related value functions[2] as follows:

$$V_n^*(b) = \max_{a \in \mathcal{A}} V_n^{*,a}(b) \tag{2.20}$$

$$V_n^{*,a}(b) = \sum_{z \in \mathcal{Z}} V_n^{*,a,z}(b) \tag{2.21}$$

$$V_n^{*,a,z}(b) = \frac{1}{|\mathcal{Z}|}\omega(b,a) + \rho\sigma(b,a,z)V_{n-1}^*(b_z^a) \ . \tag{2.22}$$

Equation 2.22 uses the identity

$expreward(b, a) = sumz1/|\mathcal{Z}|\omega(b, a)$ so that the proper immediate reward is recovered for the $V_n^{*,a}(b)$ $V_n^*(b)$ value functions.

The value function $V_n^{*,a}(\cdot)$ is the POMDP counterpart to the COMDP Equation 2.9 and has the same interpretation: the value of performing action a when there are n steps to go and then performing optimally thereafter. The value function $V_n^{*,a,z}(\cdot)$ has a slightly more complicated interpretation: it is the expected reward attributable to making observation z when action a is performed in state b when there are n decisions remaining and when the optimal actions are performed thereafter.

2.3.4 Value Function Properties

We now return to the question of how we might compute and represent POMDP policies and value functions. Recall that the main difficulty is that our state space is the infinite continuous space of probability distributions over $\mathcal{S}$. We break down the discussion into finite and infinite horizon value functions, since there are slightly different properties for each.

[2] This particular decomposition was proposed by Michael Littman.

Finite Horizon Properties

Sondik showed that the optimal finite horizon value function is *piecewise linear and convex* (PWLC) for any horizon T [117, 116]. This piecewise linear property is useful because it allows the value function to be represented using finite resources. It was this insight that allowed the development of the first exact algorithm for general finite horizon POMDPs. We will show a proof for the PWLC property below, since it will allow us to develop additional formulas that we will require in later discussions. However, before proceeding with the proof, we provide some intuition and properties of PWLC functions.

Recall that the value function is a function over $\mathcal{B}$ which is an $|\mathcal{S}| - 1$-dimensional space. Thus, in a PWLC value function, each linear segment is a hyper-plane in $|\mathcal{S}|$-space and can be represented by an $|\mathcal{S}|$-vector of coefficients. We will use γ to represent a single linear segment of a value function, $\gamma(s)$ for the s^{th} component of that vector, and Γ to represent the set of vectors or hyper-planes that comprise a PWLC value function $V(\cdot)$.

The convexity (or concavity, if you like) property means that the value function is the upper (or lower) surface of those linear value planes, where if Γ represents the value function V, then the value of an information state can be computed with

$$\begin{aligned} V(b) &= \max_{\gamma \in \Gamma} \sum_{s \in \mathcal{S}} b(s)\gamma(s) \\ &= \max_{\gamma \in \Gamma} b \cdot \gamma \quad . \end{aligned} \tag{2.23}$$

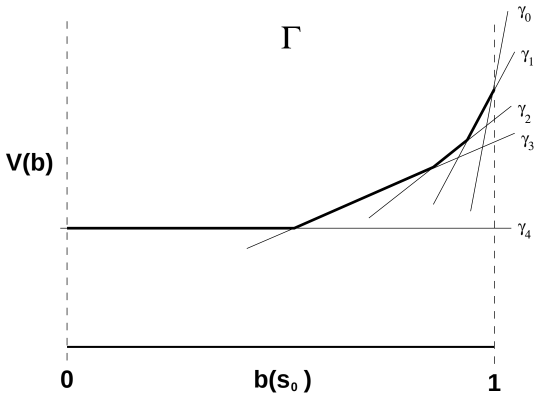


Figure 2.3: An example of a PWLC value function for a POMDP with two states.

Example As simple example of a PWLC value function, consider Figure 2.3, which is a value function for a two state POMDP where $|\Gamma| = 5$. In this figure, the state space is along the horizontal axis and values are along the vertical axis. Although the information space is specified with two probabilities, the constraint that $b(s_0) + b(s_1) = 1$ results in a 1-dimensional space and allows us to use a single number to represent any information state. In the figure, only the value for $b(s_0)$ is represented along the horizontal axis; $b(s_1) = 1 - b(s_0)$. In this figure, each linear segment of Γ is shown with a thin line and the upper surface of the value function is shown with a heavier line.

There are some useful properties of PWLC functions which we will be exploiting in the proof of the piecewise linearity and convexity of the optimal finite horizon value function. We list two of them here, though Appendix B has a complete list.

Proposition 2.3.1 *The sum of two* PWLC *functions is a* PWLC *function.*

Proposition 2.3.2 *The* max *over two* PWLC *functions is a* PWLC *function.*

Additionally, we will be needing the following fact:

Theorem 2.3.1 *If* $\forall a, z$, $V_n^{*,a,z}(\cdot)$ *is* PWLC, *then* $\forall a, V_n^{*,a}(\cdot)$ *is* PWLC *and* $V_n^*(\cdot)$ *is* PWLC.

Proof When $V_n^{*,a,z}(\cdot)$ is PWLC, using Proposition 2.3.1 and Equation 2.21 we can conclude that each of the value functions for $V_n^{*,a}(\cdot)$ are PWLC. When $V_n^{*,a}(\cdot)$ is PWLC, Using Proposition 2.3.2 and Equation 2.20 we conclude that $V_n^*(\cdot)$ is PWLC. ■

We now have all the required information to present and prove the following theorem which was first proven for the general case by Sondik [117].

Theorem 2.3.2 *For any* T, *the optimal finite horizon value function for a* POMDP *is* PWLC.

Proof The proof proceeds by induction on the horizon length. For a finite horizon problem, after the last action is taken, no more rewards are accumulated. Our induction base case is when there is a single decision remaining at $n = 1$. Here only immediate rewards matter, since the future has no value and

$$\begin{aligned} V_1^{*,a,z}(b) &= \frac{1}{|\mathcal{Z}|} \sum_{s \in \mathcal{S}} b(s) r(s,a) \quad , \quad \forall a, z \\ &= \frac{1}{|\mathcal{Z}|} b \cdot r(a) \quad . \end{aligned}$$

Therefore, each of the $V_1^{*,a,z}(\cdot)$ functions are linear and a linear function is trivially convex. Using this fact and Theorem 2.3.1, we get that $V_1^*(\cdot)$ is PWLC.

The inductive step assumes that $V_{n-1}^*(\cdot)$ is PWLC and represented with the set of vectors Γ_{n-1}. From Equation 2.23 we conclude that

$$V^*_{n-1}(b^a_z) = \max_{\gamma \in \Gamma_{n-1}} b^a_z \cdot \gamma \quad .$$

If we let

$$\chi_{n-1}(b) = \operatorname*{argmax}_{\gamma \in \Gamma_{n-1}} b \cdot \gamma \quad , \tag{2.24}$$

then we get

$$V^*_{n-1}(b^a_z) = b^a_z \cdot \chi_{n-1}(b^a_z)$$

and substituting into Equation 2.22 we get

$$V^{*,a,z}_n(b) = \frac{1}{|\mathcal{Z}|}\omega(b,a) + \rho\sigma(b,a,z)\left(b^a_z \cdot \chi_{n-1}(b^a_z)\right) \quad .$$

Substituting Equations 2.13 and 2.16 into this and canceling out the $\sigma(b,a,z)$ terms we are left with

$$\begin{aligned} V^{*,a,z}_n(b) &= \frac{1}{|\mathcal{Z}|}\sum_{s \in \mathcal{S}} b(s) r(s,a) + \rho \sum_{s \in \mathcal{S}} \sum_{s' \in \mathcal{S}} b(s)\tau(s,a,s') o(a,s',z) \chi_{n-1}(b^a_z, s') \\ &= \frac{1}{|\mathcal{Z}|}\sum_{s \in \mathcal{S}} b(s) \left[r(s,a) + \rho \sum_{s' \in \mathcal{S}} \tau(s,a,s') o(a,s',z) \chi_{n-1}(b^a_z, s') \right] \quad , \end{aligned}$$

where $\chi_{n-1}(b^a_z, s)$ is the s^{th} component of the vector $\chi_{n-1}(b^a_z)$. Letting

$$\gamma^{a,z}_n(b,s) = \frac{1}{|\mathcal{Z}|} r(s,a) + \rho \sum_{s'} \tau(s,a,s') o(a,s',z) \chi_{n-1}(b^a_z, s') \quad , \tag{2.25}$$

we have

$$V^{*,a,z}_n(b) = b \cdot \gamma^{a,z}_n(b) \quad .$$

Since there are only a finite number of possible $\gamma^{a,z}_n(b)$ vectors, as in the base case, we use Theorem 2.3.1 to conclude that $V^*_n(\cdot)$ is PWLC, which completes the induction. ■

As developed in this proof, we will define a series of vector sets, Γ_n^*, $\Gamma_n^{*,a}$ and $\Gamma_n^{*,a,z}$, each representing one of the value functions in Equations 2.20, 2.21 and 2.22 respectively, and all of which have been shown to be PWLC for all n.

Infinite Horizon Properties

Although $V_n^*(\cdot)$ is piecewise linear, and

$$\lim_{n \to \infty} ||V_n^*(\cdot) - V^*(\cdot)|| = 0 \ ,$$

this does not imply that $V^*(\cdot)$ is piecewise linear and there are POMDP problems whose optimal value functions are not piecewise linear [117]. However, there are a class of infinite horizon POMDP problems for which the optimal value function is piecewise linear. Since $V^*(\cdot)$ is PWLC and the infinite horizon problem is the limiting case for $V_n(\cdot)$, using VI and a large enough horizon, we can get as close as desired to $V^*(\cdot)$. This issue is of theoretical importance, but practically we can use a piecewise linear function to approximate any non-linear value function as closely as desired.

The property of infinite horizon POMDP policies alluded to above is called *finite transience*, which was originally defined by Sondik [117]. When a policy is finitely transient, then its value function is piecewise linear. However, there are policies with piecewise linear value functions that are not finitely transient as well as policies whose value function is not piecewise linear at all. Finitely transient policies allow for a compact representation as we will see in Appendices D and G . Unfortunately, there is no easy general way to determine when the optimal infinite horizon policy is finitely transient. However, all policies can be approximated with a finitely transient policy, which is exploited in an infinite horizon policy iteration algorithm by Sondik [118].

Despite the uncertainty about the optimal infinite horizon value function's piecewise linearity, the convexity of $V^*(\cdot)$ is preserved. This property will be exploited in some approximation schemes discussed in Chapter 5.

2.3.5 Value Iteration

Value iteration in COMDPs consists of an iteration over time, and for each time step and iteration over the states and actions, computing new values from the dynamic programming equation. Table 2.2 showed the code for performing VI in COMDPs. For POMDPs, value iteration retains the iteration over time steps; however, the continuous state space prohibits the iterations over information states. Thus, the main difficulty in implementing value iteration for POMDPs lies in the problem of computing $V_n^*(\cdot)$ from $V_{n-1}^*(\cdot)$. Chapter 3 is devoted to algorithms that perform this one-step of dynamic programming. Aside from being the basis of a VI algorithm, this one step DP step is used in many approximation schemes that are not directly based upon value iteration.

2.3.6 Policy Iteration

The two main steps in policy iteration, value determination and policy improvement, do not easily generalize from the COMDPs to the POMDPs. For an arbitrary infinite horizon policy, it is not even known if its value function is finitely representable [95], which calls into question the existence of an algorithm for the value determination step. Thus, exact policy iteration algorithms for general POMDPs do not exist and approximation methods are required.

Although we do not address policy iteration techniques in this thesis, two approximate PI algorithms, one by Sondik [117, 118] and one by Hansen [45], use the single DP step of value iteration in their policy improvement phase. The next chapter addresses the single DP step for POMDPs in detail.

2.4 Conclusions

This chapter has given the basic framework for Markov decision process formulations and solutions. These are the basic building blocks which we will use in subsequent chapters. This chapter barely scratches the surface of the theory and formalisms for MDPs and research in this area fills many volumes, though the majority of the research has been on COMDPs. Good starting references for COMDPs are Puterman's text [102] and Bertsekas' text [9], with the latter touching upon the work in POMDPs. Sondik's thesis [117] and the survey articles by Monahan [87], Lovejoy [76] and White [131] give nice overviews of both the history of the study of POMDPs, as well as the existing work in the operations research field.

Chapter 3

Exact Algorithms

All the exact algorithms for solving finite horizon POMDPs discussed in this thesis use value iteration as the basic framework and the algorithms themselves are simply different ways of computing a single dynamic programming step. For this reason, we will discuss the exact algorithms in the context of how they compute V_n^* from V_{n-1}^* or, equivalently, how they compute the set Γ_n^* from the set Γ_{n-1}^*. With an algorithm to perform this single exact DP step, embedding it in an iteration over time is all that is needed to solve a finite-horizon POMDP exactly.

The discussion in this chapter does not dwell on implementation issues or cover many of the details of the previous algorithmic approaches. Details of this sort are covered at length in earlier work [22]. Since we only discuss optimal value functions in this chapter, we will drop some of the notational clutter and let $V_n = V_n^*$ with the related functions being similarly simplified. Additionally, summations and unions for s, a and z will implicitly be defined over the state, action and observation sets respectively.

This chapter is organized into four major topics. First we will cover some concepts and issues that are common to all of the exact algorithms. We follow this with discussion of a number of exact algorithms from the algorithmic perspective. The penultimate topic presents analysis of these algorithms and we conclude this chapter with empirical results to support the analyses.

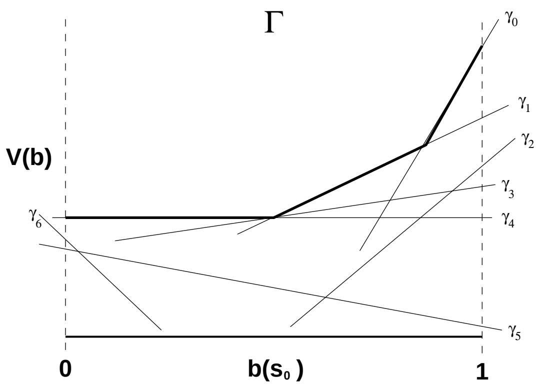


Figure 3.1: An example of a PWLC value function with useless vectors.

3.1 General Issues

We have discussed representing a piecewise linear and convex value function with a set of vectors Γ, but there are a number of issues that will continually arise concerning this representation in the algorithmic approach to the single DP step of value iteration. We will first present these common issues before moving onto the specific exact algorithms.

3.1.1 Parsimonious Representations

Given a set Γ representing a value function V as in Equation 2.23, if we construct a vector $\widetilde{\gamma}$ such that $\forall b \in \mathcal{B}$, $b \cdot \widetilde{\gamma} \leq \max_{\gamma \in \Gamma} b \cdot \gamma$, then $\Gamma \cup \{\widetilde{\gamma}\}$ will represent the exact same value function as Γ. We will refer to vectors with this property as *useless* or *dominated* vectors in the representation. As an example, Figure 3.1 shows a value function with useless vectors, γ_2, γ_3, γ_5, γ_6, in the representation.

Since there is an infinite number of vectors that could be added without changing the value function, for any PWLC value function there is an infinite number of sets that could be used as the representation. The unfortunate aspect of this is that there would not seem to be a one-to-one correspondence between a PWLC value function and its representation, nor between the size of the representation and the complexity of the value function.

In fact, it can be shown [72, 74] that any PWLC value function does have unique minimal representation. We use the term *parsimonious set*[1] when referring to the unique minimal set of vectors representing a value function. The next few sub-sections are devoted to precisely defining a parsimonious set and to presenting a reduction or *pruning* procedure for computing this set. We will see that there are some subtle issues that arise in implementing this reduction procedure.

Regions

Given a set of vectors, Γ, representing a value function over information space, we can define a partition of the information space where the partition has a finite number of elements, one for each γ in Γ. Additionally, each $\gamma \in \Gamma$ has a set or region of information states, $R(\gamma, \Gamma) \subseteq \mathcal{B}$, where it dominates, that is,

$$R(\gamma, \Gamma) = \{b | b \cdot \gamma > b \cdot \widetilde{\gamma}, \widetilde{\gamma} \in \Gamma - \{\gamma\}, b \in \mathcal{B}\} \ . \tag{3.1}$$

Note that because of the strict inequality in this definition, some information states can be in the region of none of the vectors in Γ, which makes it not quite a true partition of the information state space[2]. The set of

[1] This term is borrowed from Nevin Zhang [138].

[2] This can be made more mathematically precise, using measure theory, by eliminating consideration for regions with Lebesque measure of zero

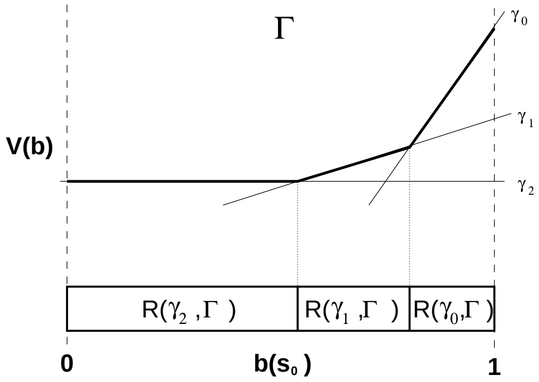


Figure 3.2: An example of the partition imposed by a PWLC value function.

points which are not in any region define the borders of the partition and are points where more than one vector gives the same maximal value. Points on these region borders will pose a problem when we need to construct vectors from information states, but we defer this discussion until Section 3.1.2. Figure 3.2 shows a value function over information space with the partition it imposes on the information space along the horizontal axis.

These regions are all that are needed for our definition:

Definition 3.1.1 *A* parsimonious *representation, Γ, of a* PWLC *value function is one where, for all $\gamma \in \Gamma$, the region $R(\gamma, \Gamma)$ is non-empty.*

Since there is a single unique hyper-plane that can be fit over any particular region, the parsimonious representation is unique [72]. Next, we develop the routines necessary for reducing a set to its parsimonious representation.

Simple Domination Checking

There is a very simple procedure, first discussed by Eagle [39], to remove some useless vectors from a non-parsimonious set $\widetilde{\Gamma}$. This procedure looks

```
dominationCheck(Γ)
    if |Γ| < 2
        then return Γ
    Γ̃ := ∅
    do
        γ := removeElement(Γ)
        if ∄γ' ∈ Γ̃ s.t. γ' ≥ γ
            then
                Γ̃ := {γ'|γ' ∈ Γ̃ , γ ≱ γ'}
                Γ̃ := Γ̃ ∪ {γ}
        end if
    until Γ = ∅
    return Γ̃
end dominationCheck
```

Table 3.1: Routine for the `dominationCheck` routine.

for vectors $\gamma \in \widetilde{\Gamma}$ where there exists some other vector $\widetilde{\gamma} \in \widetilde{\Gamma}$ such that for every $s \in \mathcal{S}$, $\gamma(s) \leq \widetilde{\gamma}(s)$. This is not guaranteed to reduce the set at all, and rarely would result in a parsimonious set all by itself, but in practice it is very effective in quickly reducing the size of the set $\widetilde{\Gamma}$.

The effectiveness of this technique lies in the fact that very little computational effort is exerted when a vector is removed from the set $\widetilde{\Gamma}$. Contrast this to the general case where determining if a vector is useless is the same as determining whether a region is empty or not, which requires setting up and solving a linear program. Table 3.1 gives a simple routine to eliminate dominated vectors from a set, though there are more efficient ways that this procedure could be implemented. In this routine the vector comparison $\gamma \geq \gamma'$ is a component-wise comparison where every component of γ is greater than or equal to the corresponding component of γ'.

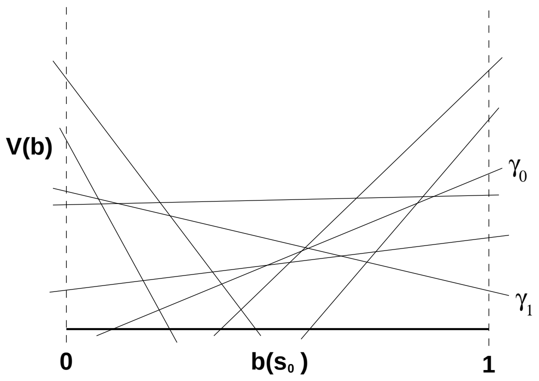


Figure 3.3: An example of a PWLC function before using the `dominationCheck` routine.

Figures 3.3 and 3.3 show a PWLC representation before and after this domination check to remove useless vectors. Notice that vectors γ_0 and γ_1 are not removed, even though they are not useful, since there is no single vector that dominates either of them.

Vector Pruning

Although the simple domination check is useful, it is not sufficient for reducing a set to its parsimonious representation. We will need a more general routine which can take an arbitrary set of vectors, $\widetilde{\Gamma}$, and reduce it to a parsimonious set, Γ, where $\widetilde{\Gamma}$ and Γ represent the same value function and $\Gamma \subseteq \widetilde{\Gamma}$.

We will first define a subroutine that explicitly encodes the definition of a region. It simply checks whether a given region is empty or not and, if it is not empty, returns an information state that lies within that region. Table 3.2 gives a routine for doing this; it takes a vector, γ and set of vectors Γ and returns **null** if the region $R(\gamma, \Gamma) = \emptyset$ or, when the region is not empty,

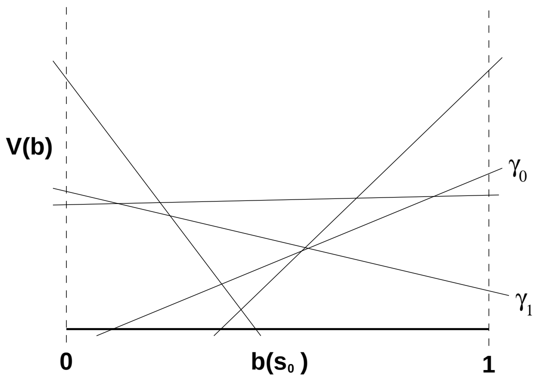


Figure 3.4: An example of a PWLC function after using the `dominationCheck` routine.

```
findRegionPoint(γ, Γ)
    L := setUpLP(γ, Γ)
    solveLP(L)
    if infeasible(L)
        then return null
    if objectiveValue(L) ≤ 0
        then return null
    return solution(L)
end findRegionPoint
```

Table 3.2: Routine for the `findRegionPoint` routine.

returns an information state b such that $b \in R(\gamma, \Gamma)$. This routine sets up and solves a linear program (LP) [137] to find such a point, where the LP is shown in Table 3.3. When the LP is infeasible or the objective function is not greater than zero, then **null** is returned, otherwise the solution point of the LP is returned.

Although this routine and LP are fairly straightforward in theory, extreme care must be used in actual implementations. Floating point compar-

variables: $\forall s \in \mathcal{S}, x(s); \varepsilon$
maximize: ε
subject to:

$$x \cdot (\gamma - \widetilde{\gamma}) \geq \varepsilon \quad , \quad \forall \widetilde{\gamma} \in \Gamma, \;\; \widetilde{\gamma} \neq \gamma$$
$$x \in \Pi(\mathcal{S})$$

Table 3.3: Linear program defined by the `setUpLP`(γ, Γ) routine.

isons, machine precision and the ranges of the vector coefficients can cause severe stability problems in the LPs, requiring a stable and robust LP solver.

The simplest approach towards finding a parsimonious set is to look at the regions $R(\gamma, \widetilde{\Gamma})$ for every vector γ. Those with non-empty regions are then added to the parsimonious set. This is the method described by Monahan [87], but is not the most efficient method. Table 3.4 gives a more efficient routine that will reduce a set of vectors to its unique parsimonious set. This pruning procedure was first proposed by Lark and White [131], though there is a subtlety involved in implementing the `bestVector` routine which is discussed below.

The algorithm works by building up the parsimonious set one vector at a time. It starts with an empty set $\widehat{\Gamma}$ and loops over the vectors in $\widetilde{\Gamma}$. At any given point in the algorithm, the set $\widehat{\Gamma}$ is a subset of the parsimonious representation Γ. Within the loop it removes a vector from $\widetilde{\Gamma}$ with the `removeElement` routine, and compares this vector against the vectors currently in $\widehat{\Gamma}$. Specifically, it looks to see if the region $R(\gamma, \widehat{\Gamma})$ is empty or not using the routine `findRegionPoint` from Table 3.2. If the region is not empty, then it will return an information state that lies within this region;

```
PRUNE(Γ̃)
    Γ̂ := ∅
    while Γ̃ ≠ ∅
        γ := removeElement(Γ̃)
        b := findRegionPoint(γ, Γ̂)
        if b ≠ null
            then
                Γ̃ := Γ̃ ∪ {γ}
                γ* := bestVector(Γ̃, b)
                Γ̃ := Γ̃ − {γ*}
                Γ̂ := Γ̂ ∪ {γ*}
        end if
    end while
    Γ := Γ̂
    return Γ
end PRUNE
```

Table 3.4: Routine for the PRUNE routine.

if the region is empty it will return **null**.

If the region is empty, then we are sure that γ is a useless vector since the vectors in the set $\widehat{\Gamma}$ already dominate it at every point, and since $\widehat{\Gamma} \subseteq \Gamma$. If the region is not empty, then we know that $\widehat{\Gamma} \subset \Gamma$, which means that there must be a vector in $\widetilde{\Gamma}$ that should be included in Γ. The subtlety here is that it is not necessarily the case that γ should be added to $\widehat{\Gamma}$; it only says that $\widehat{\Gamma}$ is not yet complete. However, the `findRegionPoint` routine provides a point, b, where the value function represented by $\widehat{\Gamma}$ is not the same as the value function represented by Γ; i.e., $\max_{\widehat{\gamma} \in \widehat{\Gamma}} b \cdot \widehat{\gamma} < \max_{\gamma \in \Gamma} b \cdot \gamma$. It is for this reason the vector γ must be put back into the $\widetilde{\Gamma}$ set before finding the best vector for the returned information state b.

In practice, the most efficient version of pruning a set to get its parsimonious representation would either preface a call to PRUNE with a call to `dominationCheck` (Table 3.1), or incorporate this call into the PRUNE routine itself.

Given a point where the value function for $\widehat{\Gamma}$ is incorrect, we can then find the proper vector from $\widetilde{\Gamma}$ to add by maximizing over the set. However, there is a very subtle issue that arises. Figure 3.5 shows the situation where we are checking the region $R(\gamma, \widehat{\Gamma})$; it shows both the current form of $\widehat{\Gamma}$ and the information state b returned by the `findRegionPoint` routine. Suppose that the true final representation, Γ, looks like Figure 3.6. We see that at point b there are two vectors that give precisely the same value, both of which would have to be in $\widetilde{\Gamma}$ when we executed `bestVector`$(\widetilde{\Gamma}, b)$. In this situation, the `bestVector` routine will be left with a choice of two equally maximal vectors to return. For the case of Figure 3.6 it could arbitrarily

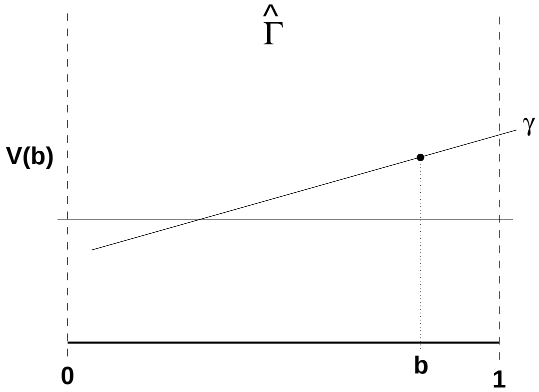


Figure 3.5: Snapshot of an example of the PRUNE routine.

decide which one to return, or we could alter the code slightly so it could return both. However, for the general case, the `bestVector` routine *cannot* be implemented in either of these ways.

Figure 3.7 shows the same true final parsimonious representation as in Figure 3.6 except we have augmented the figure with two other vectors, shown with dashed lines, that are in $\widetilde{\Gamma}$, but not in Γ. When the best vector routine executes $\texttt{bestVector}(\widetilde{\Gamma}, b)$, it will find itself with a choice of four vectors. Two of these vectors, which we term *imposters*, are not in Γ. Although there is a single point where they yield a maximal value, there is no point where they dominate every other vector, i.e., they have empty regions. The schemes of arbitrarily selecting one and returning all vectors are both wrong and will result in the PRUNE routine producing a non-parsimonious set. The next section presents the correct implementation of the `bestVector` routine that insures that no imposter vectors are returned.

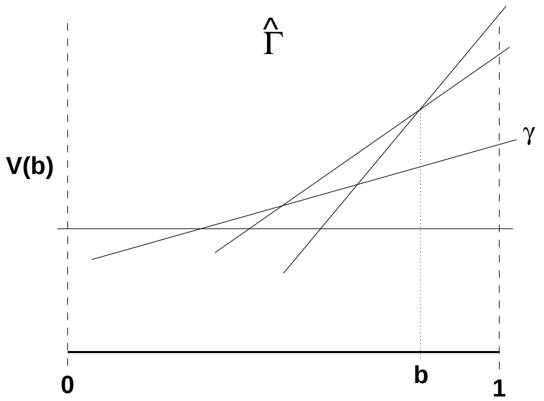


Figure 3.6: Parsimonious value function for PRUNE example.

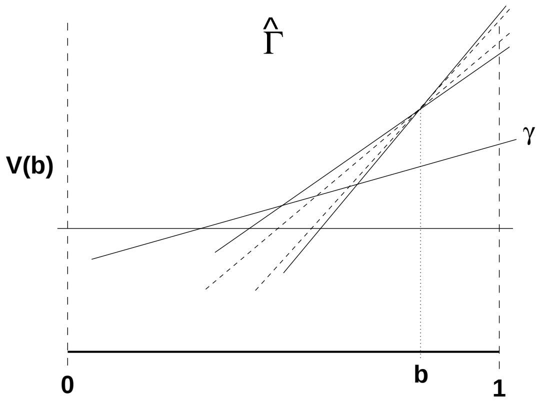


Figure 3.7: Parsimonious value function for PRUNE example augmented with imposter vectors.

Lexicographic Ordering of Vectors

As shown, there can be information points where more than one vector from Γ will provide the same maximal value, i.e., $\exists \gamma \neq \widetilde{\gamma} \in \Gamma$ such that $b \cdot \gamma = b \cdot \widetilde{\gamma}$. We would like a procedure for deterministically selecting one of these and, furthermore, to select one that is guaranteed to have a non-empty region in the final parsimonious set. To accomplish this, we define a lexicographic ordering scheme [72] over $\mathcal{S}$. The `bestVector` routine will use this ordering to deterministically decide which vector to select when more than one vector produces the same maximal value.

We define an arbitrary, though fixed, ordering over the elements in $\mathcal{S}$ such that $s \prec s'$ when state s comes before state s' in the ordering relationship.

Definition 3.1.2 *The vector γ is lexicographically greater than γ' if there exists a state s such that $\gamma(s) > \gamma'(s)$ and $\gamma(s') = \gamma'(s')$ for all $s' \prec s$.*

We will use the notation $\gamma \overset{L}{>} \gamma'$ to denote that vector γ is lexicographically greater than γ'. Table 3.5 gives a routine that returns the lexicographic maximum of two vectors. We will use this routine when faced with two vectors that yield equivalent values at an information state.

Table 3.6 shows the routine which will select, for a given information state, a vector from a set. When there is a single clear dominating vector at the point b, then this vector is returned. If one or more vectors result in the same maximal value, then the components of the vector are compared in lexicographic order until the tie is broken. It is assumed that the **for each** $s \in \mathcal{S}$ loop selects states in the fixed order defined over the set of states.

We now want to show that the `bestVector` routine always returns a

```
lexicographicMax(γ, γ̃)
    for each s ∈ 𝒮
        if γ(s) > γ̃(s)
            then return γ
        if γ(s) < γ̃(s)
            then return γ̃
    end for each s
    return γ
end lexicographicMax
```

Table 3.5: Routine for the `lexicographicMax` routine.

```
bestVector(Γ, b)
    v* := −∞
    for each γ ∈ Γ
        v := b · γ
        if v = v*
            then γ* := lexicographicMax(γ*, γ)
        if v > v*
            then
                v* := v
                γ* := γ
        end if
    end for each γ
    return γ*
end bestVector
```

Table 3.6: Routine for the `bestVector` routine using lexicographic ordering.

vector that is in the parsimonious representation of a given set. If there is only one vector $\gamma^* \in \Gamma$ where $b \cdot \gamma^* = \max_{\gamma \in \Gamma} b \cdot \gamma$, then $b \in R(\gamma^*, \Gamma)$ which means that γ^* must be in the parsimonious set by definition. We are left to show that when there is more than one vector which achieves the maximal value at b, then the lexicographic maximum choice ensures a vector in the parsimonious representation is returned.

Let Γ be any set of vectors and b be any information state. We define

$$\Lambda = \left\{ \gamma | \gamma \in \Gamma, b \cdot \gamma = \max_{\gamma \in \Gamma} b \cdot \gamma \right\} \quad ;$$

i.e., the set of all vectors that yield the same maximal value at b. We then have the following:

Theorem 3.1.1 *If there exists $\lambda^* \in \Lambda$ such that $\lambda^* \overset{L}{>} \lambda$ for all other $\lambda \in \Lambda$, then $R(\lambda^*, \Gamma)$ is non-empty.*

Proof This is proved in Appendix $G.2$ of Littman's thesis [72]. ■

Since λ^* is the vector returned by `bestVector` and it has a non-empty region, it must be part of the parsimonious representation of Γ.

3.1.2 Vector at a Point

Recall that we are only concerning ourselves with a single DP step to compute $V_n(b)$ from $V_{n-1}(b)$; in terms of representation, we are trying to compute the set Γ_n from the set Γ_{n-1}. A very important concept, which can easily be overlooked due to the myriad of formulas presented in Chapter 2, is the ease with which we can compute $V_n(b)$ for a given information state and given $V_{n-1}(\cdot)$.

Equation 2.19 (or either Equation 2.17 or 2.18) can be used to compute the value of an information state using the model parameters and a few summations over finite sets, which is relatively little effort. However, the value of a single information state is not going to be useful by itself. We will not be able to compute and store the value for each information state, which is why the finitely representable PWLC property becomes important. Instead of computing the value of an information state, we would prefer to find an element of Γ_n for an information state, which is easy to calculate by using Equation 2.25 from the proof of Theorem 2.3.2. We will find it convenient to rewrite Equation 2.25 in vector form as

$$\gamma_n^{a,z}(b) = \frac{1}{|\mathcal{Z}|} r(a) + \rho P^{a,z} \chi_{n-1}(b_z^a) \ , \tag{3.2}$$

where $r(a)$ is a column vector of immediate rewards for action a and $P^{a,z}$ is an $|\mathcal{S}| \times |\mathcal{S}|$ matrix where the element in row s and column s' is given by

$$P_{s,s'}^{a,z} = \tau(s,a,s') o(a,s',z) \ . \tag{3.3}$$

Recall that $\chi_{n-1}(b)$ is the vector from Γ_{n-1} which gives the maximal value for information state b and was defined in Equation 2.24.

As a result of Theorem 2.3.2, the $V_n^a(\cdot)$ value functions are PWLC and for a given information state, the element of Γ_n^a for that state is given by

$$\gamma_n^a(b) = \sum_z \gamma_n^{a,z}(b) \ . \tag{3.4}$$

Finally, with a γ_n^a for each action, we use

$$d_n(b) = \operatorname*{argmax}_a b \cdot \gamma_n^a(b) \tag{3.5}$$

to see which is the optimal action for b at step n, which makes the vector in Γ_n for information state b

$$\gamma_n(b) = \gamma_n^{d_n(b)}(b) \quad . \tag{3.6}$$

We note that $\gamma_n(s)$ is the s^{th} component of the vector γ_n and $\gamma_n(b)$ is the vector in Γ_n that is maximal for b. This is a notational convenience and since the arguments are of different types, the context will disambiguate the two. Also note that the s^{th} component of the vector $\gamma_n(b)$ would be $\gamma_n(b, s)$.

Therefore, for a given information state and the $n - 1^{st}$-step value function representation Γ_{n-1}, it is easy to compute its n-step value and the corresponding value hyper-plane in Γ_n. Furthermore, having computed the individual values for each action in the maximization of Equation 3.5, we have computed the optimal decision for that information point, which shows that each hyper-plane will have a specific associated action representing the policy over that hyper-plane's region.

Computing this vector, although relatively easy, presents the same problem that arose in Section 3.1.1. There may be more than one action that achieves the same maximal value, or even for the same action, there could be ties in the $\chi_{n-1}(\cdot)$ choices from the Γ_{n-1} set when constructing the individual $\gamma_n^{a,z}(b)$ vectors. When either of these situations occur, we can use the same lexicographic ordering scheme that was presented in Section 3.1.1. When we are faced with the equivalent values for the $\chi_{n-1}(b_z^a)$ vectors, we will be finding the lexicographic maximum over the Γ_{n-1} set.

3.1.3 Fixed Action Value Functions

The two exact POMDP algorithms we focus on in this chapter construct the $V_n^a(\cdot)$ value functions individually. This approach was suggested by Sondik [117], but first taken with the witness algorithm (Section 3.2) and subsequently used by the incremental pruning algorithm (Section 3.3.2). Therefore, both algorithms share the common operation of constructing Γ_n from the Γ_n^a sets.

For our representation of $V_n(\cdot)$ we could simply use $\bigcup_a \Gamma_n^a$, since for all b we have

$$\begin{aligned}\max_{\gamma \in \bigcup_a \Gamma_n^a} b \cdot \gamma &= \max_a \left[\max_{\gamma \in \Gamma_n^a} b \cdot \gamma \right] \\ &= \max_a V_n^a \\ &= V_n(b) \quad .\end{aligned}$$

Since the complexity of computing V_{n+1} will be directly related to the size of the representation of V_n, we prefer to have the parsimonious representation for V_n, i.e.,

$$\Gamma_n = \text{PRUNE}\left(\bigcup_a \Gamma_n^a \right) \quad .$$

Thus, both the witness and incremental pruning algorithms, after constructing the Γ_n^a sets, use the PRUNE routine from Section 3.1.1 to compute Γ_n.

Algorithms that attempt to construct Γ_n directly from Γ_{n-1} benefit from not having to perform this extra PRUNE operation. However, the removal of the maximization by considering the Γ_n^a sets individually often simplifies the problem and can allow techniques which would otherwise not be applicable. This issue will be discussed further in Chapter 5.

3.2 Witness Algorithm

There are a number of papers related to the witness algorithm; from when it was first proposed by Cassandra, Littman and Kaelbling [23], to elaborations on its implementation [22], and in a more formal treatment where it was proved correct [70, 74, 72]. The witness algorithm computes Γ_n^a and is based upon the idea of exploring a finite number of regions in information state space. These regions are a partition of the state space imposed by the PWLC property of the value function. Exactly how the witness algorithm does this will be elaborated upon below, but we note that some previously proposed techniques [117, 116, 26] also employ a region-based approach, though they construct Γ_n directly.

The witness algorithm is given in Table 3.7 but before discussing the algorithm in detail, we will discuss a particular relationship among the vectors that are manipulated by the algorithm and present some of its properties.

3.2.1 Neighbors

For each vector in Γ_n^a we define a set of vectors which we will call its *neighbors*, though this neighboring relationship only loosely coincides to a geometric relationship. It is neighboring vectors that will comprise the set Υ, which we call the *agenda* in the witness algorithm. Recall that we use the notation $\gamma_n^a(b)$ to refer to the vector in Γ_n^a that is maximal for b and which is constructed using Equations 3.2 and 3.4. We will use the notation without the information state argument and define its construction as

$$\begin{aligned}\gamma_n^a &= \sum_z \gamma_n^{a,z} \\ &= \sum_z \left(\frac{1}{|\mathcal{Z}|} r(a) + \rho P^{a,z} \gamma_{n-1}^z \right) \quad , \end{aligned} \tag{3.7}$$

where γ_{n-1}^z is the vector from Γ_{n-1} that was used for observation z to construct $\gamma_n^{a,z}$. Without the dependency on a particular information state, it is possible to construct vectors γ_n^a that are not in Γ_n^a; i.e., γ_n^a is not maximal for any b. We define $\overline{\Gamma}_n^a$ to be the set of all the γ_n^a we can possibly construct using Equation 3.7. We similarly define $\overline{\Gamma}_n^{a,z}$ as the set of all vectors that could be constructed from Equation 3.2. Note that $|\overline{\Gamma}_n^a| = |\Gamma_{n-1}|^{|\mathcal{Z}|}$, since each γ_n^a is constructed by selecting a vector from Γ_{n-1} for each observation.

Definition 3.2.1 *A vector ν is a* neighbor *of the vector $\gamma_n^a = \sum_z \gamma_n^{a,z}$ if*

$$\nu = \sum_{z \neq z'} \gamma_n^{a,z} + \tilde{\gamma}_n^{a,z'} \quad ,$$

for $z' \in \mathcal{Z}$, $\tilde{\gamma}_n^{a,z'} = 1/|\mathcal{Z}|r(a) + \rho P^{a,z'} \gamma_{n-1}$, for some $\gamma_{n-1} \in \Gamma_{n-1}$ and $\tilde{\gamma}_n^{a,z'} \neq \gamma_n^{a,z'}$.

The definition shows that a neighbor is any vector that differs by a single term in the summation over the observation set. Using $\overline{\Gamma}_n^a$ to represent the set of all possible γ_n^a vectors, we see that all neighbors of any vector in $\overline{\Gamma}_n^a$ are included in the set $\overline{\Gamma}_n^a$. Since there are $|\mathcal{Z}|$ possible observations and $|\Gamma_{n-1}| - 1$ possible choices for each observation, we see that each $\gamma_n^{a,z}$ vector has $|\mathcal{Z}|(|\Gamma_{n-1}| - 1)$ total neighbors. The usefulness of this neighbor relationship lies in the following theorem. We define the set of all neighbors of a vector γ as $\mathcal{N}(\gamma)$.

Theorem 3.2.1 The neighbor theorem. *For any* $\gamma_n^a \in \overline{\Gamma}_n^a$*, there exists a* $b \in \mathcal{B}$ *and a* $\widetilde{\gamma}_n^a \in \overline{\Gamma}_n^a$ *such that*

$$b \cdot \widetilde{\gamma}_n^a > b \cdot \gamma_n^a \quad ,$$

if and only if there exists a $\nu \in \mathcal{N}(\gamma_n^a)$ *where*

$$b \cdot \nu > b \cdot \gamma_n^a \quad .$$

Proof $b \cdot \nu > b \cdot \gamma_n^a$: implies $b \cdot \widetilde{\gamma}_n^a > b \cdot \gamma_n^a$ Proof in this direction is easy, since each neighbor itself is an element of $\overline{\Gamma}_n^a$.

$b \cdot \widetilde{\gamma}_n^a > b \cdot \gamma_n^a$ implies $b \cdot \nu > b \cdot \gamma_n^a$: Here we will show that such a neighbor exists by constructing it. We are given

$$\begin{aligned} b \cdot \widetilde{\gamma}_n^a &> b \cdot \gamma_n^a \\ \sum_z b \cdot \widetilde{\gamma}_n^{a,z} &> \sum_z b \cdot \gamma_n^{a,z} \quad , \end{aligned}$$

Since one summation is larger than the other, there must exist a z' such that

$$b \cdot \widetilde{\gamma}_n^{a,z'} > b \cdot \gamma_n^{a,z'} \quad .$$

Using this fact we construct ν and show that it satisfies the theorem with the derivation

$$\begin{aligned} \sum_{z \neq z'} b \cdot \gamma_n^{a,z} &= \sum_{z \neq z'} b \cdot \gamma_n^{a,z} \\ \sum_{z \neq z'} b \cdot \gamma_n^{a,z} + b \cdot \widetilde{\gamma}_n^{a,z'} &> \sum_{z \neq z'} b \cdot \gamma_n^{a,z} + b \cdot \gamma_n^{a,z'} \\ b \cdot \underbrace{\left(\sum_{z \neq z'} \gamma_n^{a,z} + \widetilde{\gamma}_n^{a,z'} \right)}_{\nu} &> b \cdot \gamma_n^a \quad , \end{aligned}$$

```
witness(Γ_{n-1}, a)
    b := any information state
    Γ̂ := {γ_n^a(b)}
    Υ := N(γ_n^a(b))
    while Υ ≠ ∅
        v := removeElement(Υ)
        if v ∈ Γ̂
            then b := null
            else b := findRegionPoint(v, Γ̂)
        end if
        if b ≠ null
            then
                Γ̂ := Γ̂ ∪ {γ_n^a(b)}
                Υ := Υ ∪ {v}
                Υ := Υ ∪ N(γ_n^a(b))
        end if
    end while
    Γ_n^a := Γ̂
    return Γ_n^a
end witness
```

Table 3.7: The witness algorithm for constructing Γ_n^a.

■

In words, Theorem 3.2.1 says that if there is a point where a better vector exists, then one of the neighbors must also be better at this point. Appendix E discusses some properties of neighbors, though it requires some of the notation and concepts from Section 3.3.

3.2.2 The Algorithm

The witness algorithm shown in Table 3.7 begins by selecting any information state, then constructing the maximal vector for it as well as all of its neighbors. It is important that the $\gamma_n^a(b)$ vector be constructed using the lexicographic ordering scheme (Section 3.1.2) for the final result to be a parsimonious set. The set $\widehat{\Gamma}$ is used to store the vectors that comprise Γ_n^a as they are discovered by the algorithm; at all times $\widehat{\Gamma} \subseteq \Gamma_n^a$ and the value function, $\widehat{V}$, represented by $\widehat{\Gamma}$ is an underestimate so that: $\forall b, \widehat{V}(b) \leq V_n^a(b)$. The neighbors are put into a set of agenda items Υ, which serves as the basis for the iteration.

The loop removes neighbor vectors from Υ one at a time and terminates when the set is empty. After a neighbor υ is removed from Υ, if it is not already in $\widehat{\Gamma}$, we determine whether or not an information state exists where this neighbor would be better than all the vectors in our current $\widehat{\Gamma}$ set. Formally, we are checking the region $R(\upsilon, \widehat{\Gamma})$ and returning an information state in the region if one exists.

If the region is empty or the υ is already in $\widehat{\Gamma}$, then we discard υ and select a new one from our agenda. When the region is not empty, using any point b from within the region, we construct the maximal vector for that point (using Equation 3.4) and add it to $\widehat{\Gamma}$. We then put υ back into the set Υ and add all the neighbors of the new vector $\gamma_n^a(b)$ to the agenda. Putting υ back into Υ is an important step in ensuring the algorithm's correctness. Putting υ back into Υ is important because although we have found that υ is not maximal at b, we have not demonstrated that it is useless over the entire region.

The intuition behind why this algorithm works lies in Theorem 3.2.1. No matter what vectors are in $\widehat{\Gamma}$ at any point in time, if there is a vector in $\overline{\Gamma}_n^a$ which would give a higher value at a belief point b, then one of the neighbors ν of the vector $\widehat{\gamma} \in \widehat{\Gamma}$ that is currently best at b also gives a larger value at this point and thus has a non-empty region $R(\nu, \widehat{\Gamma})$. Since we ensure that all neighbors are checked, we will not miss any vector in $\overline{\Gamma}_n^a$ that belongs in Γ_n^a. The actual proof that this algorithm is correct becomes a little more complex, since we have to ensure it works for any particular order the vectors are added and any particular order that the neighbors are checked.

Theorem 3.2.2 *The witness algorithm correctly constructs the parsimonious representation, Γ_n^a, for the value function V_n^a.*

Proof The proof breaks down into four steps:

1. If γ is added to $\widehat{\Gamma}$, then $\gamma \in \Gamma_n^a$.
2. If γ is added to $\widehat{\Gamma}$, then $\gamma \notin \widehat{\Gamma}$ just before it is added.
3. The algorithm terminates in a finite number of iterations.
4. When the algorithm terminates, $\widehat{\Gamma} = \Gamma_n^a$

Step 1: We first show that any vector added to $\widehat{\Gamma}$ must be a vector in Γ_n^a. Any vector added to $\widehat{\Gamma}$ is constructed directly from an information state using Equation 3.4 and ties are broken using the lexicographic ordering scheme of Section 3.1.1. The fact that $\gamma_n^a \in \Gamma_n^a$ follows from Theorem 3.1.1 which assures that we select a vector with a non-empty region in Γ_n^a.

Step 2: Next we show that it never adds the same vector twice to $\widehat{\Gamma}$. We prove this by contradiction. Assume that `findRegionPoint`$(v, \widehat{\Gamma})$ routine returns a b such that $\gamma_n^a(b) \in \widehat{\Gamma}$, then we know that $b \cdot \gamma_n^a(b) \geq b \cdot \overline{\gamma}, \forall \overline{\gamma} \in \overline{\Gamma}_n^a$.

It must also be the case that $b \cdot v > b \cdot \widehat{\gamma}, \forall \widehat{\gamma} \in \widehat{\Gamma}$, since the algorithm ensures that all v considered are not in $\widehat{\Gamma}$. Because $v \in \overline{\Gamma}_n^a$ and $\gamma_n^a(b) \in \widehat{\Gamma}$ we have a contradiction, since we cannot have $b \cdot \gamma_n^a(b) \geq b \cdot v$ and $b \cdot v > b \cdot \gamma_n^a(b)$ simultaneously.

Step 3: We now show that the algorithm terminates. Each iteration through the loop either removes a vector from Υ or adds a vector to $\widehat{\Gamma}$ and its neighbors to Υ. We know that Γ_n^a is finite and we only add vectors to $\widehat{\Gamma}$ that are in Γ_n^a. Since we never add the same vector twice to $\widehat{\Gamma}$, we only attempt to add a vector to $\widehat{\Gamma}$ a finite number of times. Since each vector only has a finite number of neighbors, we only add a finite number of neighbors to Υ. Therefore, the loop can only execute a finite number of times.

Step 4: Finally, we show that the final $\widehat{\Gamma} = \Gamma_n^a$ with a proof by contradiction. Assume that the algorithm terminates with $\widehat{\Gamma} \neq \Gamma_n^a$, then there must exist an information state b and a vector $\overline{\gamma} \in \overline{\Gamma}_n^a$ where $b \cdot \overline{\gamma} > b \cdot \widehat{\gamma}$, where $\widehat{\gamma} = \operatorname{argmax}_{\gamma \in \widehat{\Gamma}} b \cdot \gamma$. Using this and Theorem 3.2.1, we know that there must be a neighbor, ν, of $\widehat{\gamma}$ such that $b \cdot \nu > b \cdot \widehat{\gamma}$. However, we have added all neighbors of $\widehat{\gamma}$ to Υ when we added $\widehat{\gamma}$ to $\widehat{\Gamma}$ and only removed ν from Υ when $R(\nu, \widehat{\Gamma}) = \emptyset$. But if $R(\nu, \widehat{\Gamma}) = \emptyset$ then $\forall b$ we must have $b \cdot \nu \leq b \cdot \widehat{\gamma}$ for all $\widehat{\gamma} \in \widehat{\Gamma}$. This contradicts the statement that $b \cdot \nu > b \cdot \widehat{\gamma}$. ∎

The proof that the witness algorithm correctly computes Γ_n^a first appeared elsewhere [72]. Appendix F.2 shows an example of how this algorithm works on the simple baseball problem which was introduced in the previous chapters. The witness algorithm has some appealing theoretical guarantees which are discussed further in Section 4.1.4. The main result is that constructing the set Γ_n^a can be done in time polynomial in $\mathcal{S}$, $\mathcal{A}$, $\mathcal{Z}$, Γ_{n-1} and Γ_n^a.

3.2.3 Witness Optimizations

As presented in the previous section, the witness algorithm is not in its most efficient form. This section briefly discusses some of the optimizations that can be incorporated into this algorithm to help reduce the amount of computation required. Some of these optimizations are applicable for other algorithms and are discussed further in Section 4.7 in the context of analyzing the algorithms.

Initializing $\widehat{\Gamma}$ The witness algorithm initializes $\widehat{\Gamma}$ with a single vector and all other vectors added to $\widehat{\Gamma}$ will be the result of solving an LP. A simple optimization is to check every vertex of the information space simplex and initialize $\widehat{\Gamma}$ to be each vector computed at these points. If $|\Gamma_n^a| > 1$, then we are ensured of finding at least two vectors this way, and in practice it can be considerably more. A slight generalization of this idea is to initialize $\widehat{\Gamma}$ with vectors generated from some set of random information states. This isn't guaranteed to find more than one vector, but could save a reasonable amount of time in practice.

Discarded Neighbors After a neighbor ν is removed from Υ, it is possible that a vector γ subsequently added to $\widehat{\Gamma}$ will result in ν being added back into Υ; i.e., $\nu \in \mathcal{N}(\gamma)$. Because we only removed ν when $R(\nu, \widehat{\Gamma})$ was empty, by adding more vectors to $\widehat{\Gamma}$ it is not possible for this region to become non-empty. A simple optimization keeps tracks of neighbors that have been removed from Υ to ensure they are not again added to Υ.

Domination Checking The simple domination check discussed in Section 3.1.1 can be used in conjunction with the witness algorithm and, on a small sample of problems, has reduced the computation time by as much as 50%. Its incorporating into the witness algorithm checks each v before the `findRegionPoint` routine. Applied to this case, it simply looks to see if there is a vector in $\widehat{\Gamma}$ which already dominates v. If v is already dominated by $\widehat{\Gamma}$, we proceed as if the `findRegionPoint` routine returned **null**. An alternative is to check an item against $\widehat{\Gamma}$ before it is put into Υ rather than when it is taken out, which leads to the idea for the next optimization.

Redundant Neighbors In the description of the witness algorithm when adding γ to $\widehat{\Gamma}$ we are required to add all of its neighbors, which is sufficient for correctness but not entirely necessary. All the witness algorithm really requires is that for every $b \in R(\gamma, \widehat{\Gamma})$, where $b \cdot \gamma < V_n^a(b)$, that we add a vector γ' to Υ such that $b \cdot \gamma' > b \cdot \gamma$. Adding all neighbors guarantees this by Theorem 3.2.1, however, for a given neighbor ν, if for all those information states there is another vector γ', either in Υ or in $\mathcal{N}(\gamma)$, where $b \cdot \gamma' \geq b \cdot \nu$ then adding ν is not necessary, since γ' is just as sufficient as γ in finding these points. For each $\nu \in \mathcal{N}(\gamma)$, we can compare it to the vectors currently in Υ or with the other vectors in $\mathcal{N}(\gamma)$, using either the simple domination check, or more completely with an LP. The former is likely to save time, while the latter may only have diminishing returns. Avoiding putting an item in Υ is done for the sake of saving an LP, so the requirement of an LP to decide whether to put the item into Υ may be of little help. If the LP determines that the item must be added, then we have done two LPs where normally one would have sufficed.

Region Adjacency Information There is an optimization that Smallwood and indexSondik, Edward J.Sondik [116] propose for their one-pass algorithm which uses information about the adjacency of regions to reduce the amount of computation that is required. Givan [43] has pointed out that this same idea could be applied to other algorithms, including witness, which search in information space.

Suppose we find a vector $\gamma_n^a = \sum_z \gamma_n^{a,z}$ and we are considering adding all $|\mathcal{Z}||\Gamma_{n-1}|$ vectors of $\mathcal{N}(\gamma_n^a)$ to Υ. The purpose of this is to ensure that if $R(\gamma_n^a, \widehat{\Gamma})$ is too large, then we will have a vector in Υ that will lead to a witness point. The observation needed here is that if the region is too large, then there must be some point on the border of its true region, $R(\gamma_n^a, \Gamma_n^a)$, that would be a witness point. For each observation, z, instead of considering all $|\Gamma_{n-1}|$ neighbors in $\Gamma_n^{a,z}$, it suffices to limit ourselves to those $\widetilde{\gamma}_n^{a,z} \in \Gamma_n^{a,z}$ where $R(\widetilde{\gamma}_n^{a,z}, \Gamma_n^{a,z})$ is adjacent to $R(\gamma_n^{a,z}, \Gamma_n^{a,z})$.

To implement this optimization requires generating and storing the adjacency information for each $\Gamma_n^{a,z}$. This is not as bad as it sounds since this adjacency information is essentially derived from the adjacency information from Γ_{n-1}. For Smallwood and Sondik's algorithm, this adjacency information must be computed anyway, so there is no extra work involved. Unfortunately, the witness algorithm does not generate regions in a way that immediately gives the adjacency information. Therefore, this information must be explicitly computed which leads to the potential problems with this optimization. We must ensure that the time spent computing adjacency information does not exceed the time saved from eliminating vectors that are added to Υ. This will be directly related to the relative adjacency of regions in high dimensional space for typical POMDP problems, which is mostly unknown at this point, though Zhang [139] has some preliminary results which point to this relationship being relatively sparse.

3.3 Incremental Pruning Algorithms

The incremental pruning algorithm was first proposed by Zhang and Liu [140] and was subsequently analyzed, compared and improved [24]. Like witness, it breaks down the problem into constructing the Γ_n^a sets individually. Unlike the witness algorithm, this method does not search in regions of the state space; instead, it considers constructing each possible $\gamma_n^a \in \Gamma_n^a$. However, it constructs each vector in an incremental fashion which reduces the computational complexity significantly.

3.3.1 Batch Enumeration

To understand to basic approach of the incremental pruning algorithm, it helps to first discuss the batch enumeration algorithm as presented by Monahan [87][3], since it is conceptually the simplest of all the exact algorithms.

The witness algorithm, and the algorithms to be discussed in Section 3.4, search in information space for a set of points that are able to yield the full parsimonious representation of the value function. This approach required defining regions in the information space and performing some sort of search to find points in other regions. The alternative approach comes from looking at Equation 3.2 in a slightly different manner. We repeat this equation here for convenience:

$$\gamma_n^{a,z}(b) = \frac{1}{|\mathcal{Z}|} r(a) + \rho P^{a,z} \chi_{n-1}(b_z^a) \quad ,$$

The only function that the specific information state plays in this formula

[3]Sondik [117] actually proposes such a scheme, but never presents it as an algorithm in its own right. Curiously, Monahan presents this algorithm under the guise of Sondik's one-pass algorithm.

is in the $\chi_{n-1}(\cdot)$ term, which does nothing other than select a vector from Γ_{n-1}. Regardless of the specific information state, there are only $|\Gamma_{n-1}|$ possible values for $\chi_{n-1}(b_z^a)$, which means that there are only as many possible $\gamma_n^{a,z}$ vectors for a given action a and observation z. We define

$$\overline{\Gamma}_n^{a,z} = \left\{ \frac{1}{|\mathcal{Z}|} r(a) + \rho P^{a,z} \gamma_{n-1} | \gamma_{n-1} \in \Gamma_{n-1} \right\} \quad ,$$

which is the set of all possible $\gamma_n^{a,z}$ vectors. Note that $|\overline{\Gamma}_n^{a,z}| = |\Gamma_{n-1}|$.

Since $\gamma_n^a = \sum_z \gamma_n^{a,z}$, we define $\overline{\Gamma}_n^a$ to be the set of vectors obtained for all ways of selecting a vector from $\overline{\Gamma}_n^{a,z}$, for each z. Appendix B defines the *cross-sum* operator and some of its properties. The cross-sum operator, which we denote with $\oplus$, operates on two sets of vectors, A and B, and produces a set of vectors that consists of all pairwise additions of vectors from both sets. Using this operator we can define $\overline{\Gamma}_n^a$ more precisely as

$$\overline{\Gamma}_n^a = \bigoplus_z \overline{\Gamma}_n^{a,z} \quad . \tag{3.8}$$

Since all possible combinations of the $\gamma_n^{a,z}$ vectors are in $\overline{\Gamma}_n^a$, the maximal vector for any information state point will also be there; i.e., $V_n^a(b) = \max_{\overline{\gamma} \in \overline{\Gamma}_n^a} b \cdot \overline{\gamma}$.

This is exactly the approach of the enumeration-type algorithms; they ignore information states and simply generate every possible vector. To complete the enumeration from above, we need to repeat this for each action and so we define

$$\overline{\Gamma}_n = \bigcup_a \overline{\Gamma}_n^a \quad ,$$

which is the complete enumeration of all possible vectors that could be in Γ_n.

This complete, or batch, enumeration algorithm as presented in Monahan [87] gives a simple linear programming scheme for reducing the $\overline{\Gamma}_n$ set to its parsimonious representation. Subsequent improvements to this batch enumerative scheme were merely ways to do this reduction (or pruning) phase more efficiently. Eagle [39] added the domination checks discussed in Section 3.1.1 and Lark [131] devised a more efficient linear programming approach which is the basis of the PRUNE routine of Section 3.1.1.

Thus the batch enumeration algorithm can be summarized succinctly as:

$$\Gamma_n = \text{PRUNE}\left(\bigcup_a \bigoplus_z \overline{\Gamma}_n^{a,z}\right) .$$

This batch enumeration scheme has the computation complexity of being exponential in the size of $\mathcal{Z}$ regardless of the size of Γ_n. For POMDP problems with more than 3 or 4 observations, this technique is impractical. However, we will see in the sections to follow that this same enumeration idea can be performed incrementally to yield a much more effective algorithm.

3.3.2 Incremental Enumeration

As discussed in Section 3.1.3, the incremental pruning (IP) algorithm concerns itself with constructing the Γ_n^a sets. Since $\Gamma_n^a = \text{PRUNE}(\overline{\Gamma}_n^a)$, from Equation 3.8 we have

$$\Gamma_n^a = \text{PRUNE}\left(\bigoplus_z \overline{\Gamma}_n^{a,z}\right) ,$$

which shows a batch enumeration-type method for computing the $\overline{\Gamma}_n^a$ sets followed by a pruning phase to yield Γ_n^a.

One improvement that can be made uses some properties of the cross-sum operator (see Appendix B) and moves the PRUNE routine inside the cross-sum operator yielding

$$\begin{aligned}\Gamma_n^a &= \text{PRUNE}\left(\bigoplus_z \text{PRUNE}\left(\overline{\Gamma}_n^{a,z}\right)\right)\\ &= \text{PRUNE}\left(\bigoplus_z \Gamma_n^{a,z}\right) \quad . \end{aligned} \tag{3.9}$$

Note that although we can safely move the pruning step inside the cross-sum operator, it is still required on the outside. The fact that any vector in $\overline{\Gamma}_n^{a,z}$ that gets pruned never needs to be considered anywhere else can actually be exploited in other algorithms (see Section 3.2.3). For the remainder of this section, we will always assume that we prune the $\overline{\Gamma}_n^{a,z}$ sets and always deal directly with $\Gamma_n^{a,z}$.

The incremental pruning algorithm's insight is the simple, yet elegant idea of interleaving the PRUNE routines with the cross-sum operators in Equation 3.9. This is possible due to properties of the cross-sum operator. Here we show the progression, starting with Equation 3.9

$$\begin{aligned}\Gamma_n^a &= \text{PRUNE}\left(\bigoplus_z \Gamma_n^{a,z}\right)\\ &= \text{PRUNE}\left(\Gamma_n^{a,0} \oplus \Gamma_n^{a,1} \oplus \Gamma_n^{a,2} \oplus \ldots \oplus \Gamma_n^{a,|\mathcal{Z}|-1}\right)\\ &= \text{PRUNE}\left(\ldots \text{PRUNE}\left(\text{PRUNE}\left(\Gamma_n^{a,0} \oplus \Gamma_n^{a,1}\right) \oplus \Gamma_n^{a,2}\right) \ldots \oplus \Gamma_n^{a,|\mathcal{Z}|-1}\right) \quad . \end{aligned} \tag{3.10}$$

Equation 3.10 is essentially the incremental pruning algorithm. The algorithm is summarized in Table 3.8 and an example of it working on a simple problem is given in Appendix F.1.

The algorithm in Table 3.8 is fairly straightforward, but some explanation will be useful. The elements of the set Ψ are sets of vectors, so Ψ merely

```
incrementalPrune(Γ_{n-1}, a)
    Ψ := ⋃_z {Γ_n^{a,z}}
    while |Ψ| > 1
        A := removeElement( Ψ )
        B := removeElement( Ψ )
        D := PRUNE(A ⊕ B)
        Ψ := Ψ ⋃ {D}
    end while
    return Ψ
end incrementalPrune
```

Table 3.8: Routine for the incremental pruning algorithm.

serves to hold the $\Gamma_n^{a,z}$ sets and the intermediate results of the cross-sum operations. The routine `removeElement()` simply extracts one of the vector sets from Ψ. Note that a routine for the batch pruning algorithm is nearly identical to the routine shown in Table 3.8. The only change is to replace the line $D := \text{PRUNE}(A \oplus B)$ with $D := A \oplus B$ and add $\Psi := \text{PRUNE}(\Psi)$ just before returning Ψ.

Although Equation 3.10 shows one particular grouping of the sets in the cross-sum, there are many such groupings. To accomplish any other grouping requires no changes in the algorithm, just a simple change in the `removeElement` routine. In general no one grouping is preferable to any other, but for given assumptions about the structure of the solution, some are preferable to others. This issue is explored further in Section 4.4.1.

An interesting aspect of the algorithm given in Table 3.8 is that by removing the PRUNE calls, the algorithm becomes the batch enumeration algorithm for constructing the Γ_n^a sets.

3.3.3 Generalized Incremental Pruning

The generalization of the incremental pruning algorithm discussed in this section grew out of trying to refine the incremental pruning algorithm and was first presented by Cassandra, Littman and Zhang [24]. The basic structure of the incremental algorithm is the same; the main difference lies in the way the individual PRUNE$(A \oplus B)$ operations are performed.

In the normal application of the PRUNE operation, there is nothing but a set of vectors to work on. However, in computing PRUNE$(A \oplus B)$ we have some intimate knowledge about the way in which the set being pruned is constructed. The generalized incremental pruning algorithm (GIP) is a technique that can exploit this knowledge.

Before presenting the full GIP algorithm, we present one motivating example of the approach. Consider the simple problem of computing PRUNE$(A \oplus B)$ where A and B are both parsimonious representations. We are looking for the parsimonious representation of the cross-sum of these two sets, which we will refer to as D. We will use $\overline{D} = A \oplus B$, which is the full, possibly non-parsimonious cross-sum. Recall, that to be parsimonious means that for every vector, δ in the set, D, $R(\delta, D) \neq \emptyset$.

Let us first focus on a single vector, $\alpha + \beta$ from the cross-sum. If we did the full cross-sum and then pruned the set, it is possible that to determine whether or not $R(\alpha + \beta, \overline{D}) = \emptyset$ we will have to compare it against all the other $|\overline{D}| - 1$ vectors. In the pruning routine, this will translate into an LP with roughly $|\overline{D}| = |A||B|$ constraints. We note that comparing it to this many is often not necessary with the PRUNE operation, but in the worst case

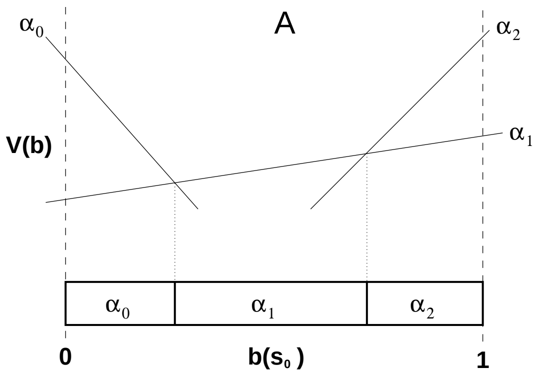


Figure 3.8: Value function and partition for PWLC set A.

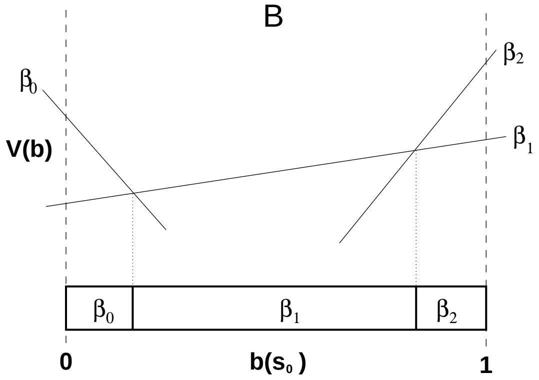


Figure 3.9: Value function and partition for PWLC set B.

this might happen.

To reduce the size of the LP, we can exploit the geometry of this problem. Figures 3.8 and 3.9 shows an example of two sets of vectors for A and B. The division on the horizontal axis shows the partitions that are imposed by these sets with the symbol for the vector in each region. Figure 3.10 shows just the two sets' partitions aligned.

Looking at the region, $R(\alpha_0, A)$, imposed by the vector α_0 in Figure 3.8,

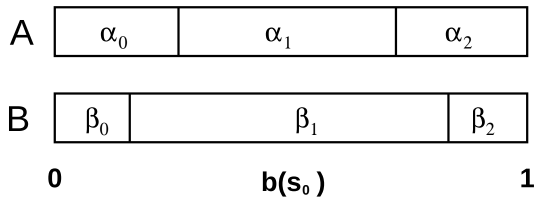


Figure 3.10: Partitions for PWLC sets A and B.

consider what possible combinations of vectors will yield a maximal vector within this region. The first thing that becomes clear is that over this region, for all β in B no $\alpha_1+\beta$ or $\alpha_2+\beta$ will be larger than $\alpha_0+\beta$ in $R(\alpha_0, A)$. The next thing that becomes clear is that $\alpha_0+\beta_2$ cannot be maximal anywhere over this region either.

In two dimensions, it is visually easy to see which combinations from A and B will comprise D: any two vectors whose regions overlap. Figure 3.11 shows the final partitions imposed by $A \oplus B$; notice how the region boundaries of $A \oplus B$ are defined by the region boundaries of A and B. This can also be made geometrically precise. Therefore, if we want to determine if $R(\alpha+\beta, \overline{D}) = \emptyset$, it is equivalent to check the whether $R(\alpha, A) \cap R(\beta, B) = \emptyset$. In rough terms of linear program size, the former will have $|A||B|$ constraints and the latter will have $|A|+|B|$ constraints. More precise and thorough analysis are provided in Section 4.4.5.

Although this provides the main motivation for the GIP algorithm, it is not entirely accurate and hides many of the more subtle aspects. First, the PRUNE routine is more efficient than always checking $R(\alpha+\beta, \overline{D}) = \emptyset$ for each vector. Second, this approach always requires $|A|+|B|$ constraints and the PRUNE routine will do many LPs which are smaller than this, since it

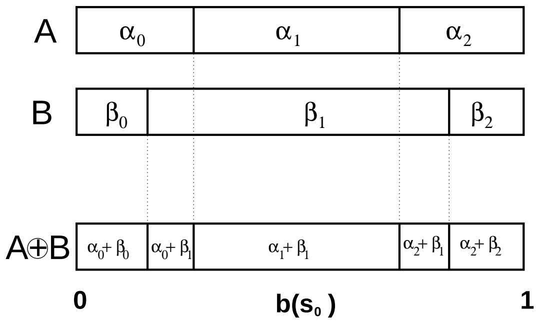


Figure 3.11: The final partition for $A \oplus B$ and its relationship to the initial partitions of A and B.

processes the vectors in a more efficient manner.

The real contribution of the GIP algorithm is that it is able to incorporate the same insights from the PRUNE routine into this geometric intersection region-based view.

We will be using the following notation in the presentation of the GIP algorithm:

$$\overline{D} = A \oplus B$$

$$D = \text{PRUNE}(\overline{D})$$

$$\widehat{D} \subseteq D$$

$$\widehat{D}_\alpha = \{\alpha + \widehat{\beta} | \widehat{\beta} \in B, (\alpha + \widehat{\beta}) \in \widehat{D}\}$$

$$\widehat{D}_\beta = \{\widehat{\alpha} + \beta | \widehat{\alpha} \in A, (\widehat{\alpha} + \beta) \in \widehat{D}\} \ .$$

The GIP algorithm uses the same routine as IP except that the PRUNE$(A \oplus B)$ line is replaced with a call to the routine genCrossSum(A, B). The routine for this new cross-sum routine is given in Table 3.9. It is nearly identical to the PRUNE routine (Table 3.4) with the exception of some notation and

```
genCrossSum(A, B)
    D̂ := ∅
    D̃ := A ⊕ B
    while D̃ ≠ ∅
        γ := removeElement(D̃)
        b := findRegionPoint(γ, Δ)
        if b ≠ null
            then
                D̃ := D̃ ∪ {γ}
                γ := bestVector(D̃, b)
                D̃ := D̃ − {γ}
                D̂ := D̂ ∪ {γ}
        end if
    end while
    D := D̂
    return D
end genCrossSum
```

Table 3.9: Routine for the GIP cross-sum `genCrossSum`.

the line $b :=$ `findRegionPoint`(γ, Δ). We call this the *generalized cross-sum* (GCS) algorithm. Note that in the algorithm $\widetilde{D}$ itself is a subset of $\overline{D}$ which serves as the set holding those vectors in $\overline{D}$ which have not been checked.

At any given iteration of the **while** loop, a vector $\gamma = \alpha + \beta$ is selected. It is important to note that the implementation of this algorithm now requires keeping track of how each γ vector was constructed. Any of the following sets can be used in the call to `findRegionPoint`:

1. $\Delta = \overline{D}$

2. $\Delta = (\{\alpha\} \oplus B) \cup (A \oplus \{\beta\})$

3. $\Delta = \widehat{D}$

4. $\Delta = \widehat{D}_\alpha \cup (A \oplus \{\beta\})$

5. $\Delta = (\{\alpha\} \oplus B) \cup \widehat{D}_\beta$.

Using the first set is equivalent to using Monahan's original pruning algorithm on the cross-sum set. The second case corresponds to the motivating example we gave for GIP and effectively is computing the intersection of $R(\alpha, A)$ and $R(\beta, B)$. The third case corresponds to the original IP algorithm where the PRUNE routine is used to prune the set. The last two cases combine the ideas of the second and third cases. The best GIP algorithm would choose whichever Δ was smallest when a given γ is checked. In Chapter 4 we will analyze and empirically evaluate a slightly less efficient form of GIP which always chooses one of the last two sets. Nonetheless we will find this version to be more effective than the normal IP algorithm.

The correctness of using either of the last two cases is not intuitively obvious from simply looking at their definitions. What these cases amount to is a PRUNE operation with some additional constraints. For example, suppose we want to find all the vectors in B which yield useful vectors when combined with the vector α from A. This corresponds to using the fourth set from above. From our GIP motivating example, we know that we only need to examine the set B over the region $R(\alpha, A)$. The example then searched the entire set B, one vector at a time, comparing it to all the other vectors in B. However, using the same insight that is used in the PRUNE routine, we can gradually build up a set of the useful $\beta \in B$ and compare each subsequent vector only to the current approximation instead of the entire set.

This effectively does a PRUNE operation on the set B subject to the additional constraints imposed by $R(\alpha, A)$. Given a particular vector, $\alpha + \beta$, to check in the GIP algorithm, we see that $R(\alpha, A) = R(\alpha+\beta, A \oplus \{\beta\})$. Additionally, if we are simply filtering the set B (with additional constraints), then the current approximation $\widehat{D}$ is comprised of some number of vectors from B and we iteratively compare all the vectors in B to $\widehat{D}$. However, if in this comparison, we simply shifted all the vectors in B upwards, then the results will not change, since shifting all the vectors simulataneously does not change their region boundaries. If we view α as the amount we shift then the filtering algorithm can be viewed as operating on the set $\{\alpha\} \oplus B$, where the current approximation $\widehat{D}$ has components $\alpha + \widehat{\beta}$ for the $\widehat{\beta}$ thus far found[4]. This is precisely the notion that the set $\widehat{D}_\alpha$ captures, and we see that $R(\beta, \widehat{D}) = R(\alpha + \beta, \widehat{D}_\alpha)$.

In the discussion above, we would refer to A as the *resticting set* since it defines the sub-regions over which we look for useful vectors from B. In Sections 4.3.2 and 4.4.4, we will discuss in more detail a variation of the GIP algorithm which *always* chooses one of these last two cases.

The following two theorems show that all of the sets above are valid to use to construct the parsimonious representation of the cross-sum.

Theorem 3.3.1 *If $R(\gamma, \Delta) = \emptyset$ then $\gamma \notin D$.*

Proof To prove this theorem, we use the fact that for all the cases we have

[4]Although adding an arbitrary vector to this set does not shift the vectors upwards equally at all points, the shifting does not change the partition boundaries, which is all that matters for this search.

$\Delta \subseteq \overline{D}$. If the region is empty over a subset of $\overline{D}$, then it certainly will be empty over the entire $\overline{D}$ set. By definition, $\gamma \in D$ if and only if $R(\gamma, \overline{D}) \neq \emptyset$. ■

This theorem shows that all vectors discarded are useless vectors. Next we show that all vectors added to $\widehat{D}$ are in D.

Theorem 3.3.2 *If $\exists b \in R(\gamma, \Delta)$ then $\exists \gamma^* \in \widetilde{D}$ such that $b \in R(\gamma^*, D)$.*

Proof Let $\widehat{\gamma} = \widehat{\alpha} + \widehat{\beta} = \text{argmax}_{\gamma \in \widehat{D}}\, b \cdot \gamma$, which is simply the best vector in our current approximation $\widehat{D}$ at information state b. Because we are checking the vector $\gamma = \alpha + \beta$, we know that $\gamma \notin \widehat{D}$, since every time we add a vector to $\widehat{D}$ we remove it from $\widetilde{D}$. Let $\gamma^* = \text{argmax}_{\overline{\gamma} \in \overline{D}}\, b \cdot \overline{\gamma}$, which is the best vector at b among all possible vectors in $A \oplus B$, then we know that $b \cdot \gamma^* \geq b \cdot \gamma$ for any other γ. If we can show that $b \cdot \gamma^* > b \cdot \widehat{\gamma}$ then we know that $\gamma^* \notin \widehat{D}$ and could not have been removed from $\widetilde{D}$ since $R(\gamma^*, \Delta) \neq \emptyset$ for any subset Δ of $\overline{D}$.

We now show that $b \cdot \gamma^* > b \cdot \widehat{\gamma}$ for each Δ.

1. $\Delta = \overline{D}$

   Since $b \in R(\gamma, \overline{D})$, this implies $\gamma = \gamma^*$ and since γ is not in $\widehat{D}$, $b \cdot \gamma^* > b \cdot \widehat{\gamma}$.

2. $\Delta = (\{\alpha\} \oplus B) \cup (A \oplus \{\beta\})$

   By nature of this set we have $\forall \overline{\beta} \neq \beta \in B$

$$b \cdot (\alpha + \beta) > b \cdot (\alpha + \overline{\beta})$$

$$b \cdot \beta > b \cdot \overline{\beta}$$

and $\forall \overline{\alpha} \neq \alpha \in A$

$$b \cdot (\alpha + \beta) > b \cdot (\overline{\alpha} + \beta)$$

$$b \cdot \alpha > b \cdot \overline{\alpha} \ .$$

Therefore, $b \cdot (\alpha + \beta) > b \cdot (\overline{\alpha} + \overline{\beta})$ for all other vectors $(\overline{\alpha} + \overline{\beta}) \in A \oplus B$ and $\gamma = \gamma^*$. As in the first case, showing $\gamma = \gamma^*$ implies that $b \cdot \gamma^* > b \cdot \widehat{\gamma}$.

3. $\Delta = \widehat{D}$

   Since $\gamma \notin \widehat{D}$, if $b \in R(\gamma, \widehat{D})$ then $b \cdot \gamma > b \cdot \widehat{\gamma}$. Since $b \cdot \gamma^* \geq b \cdot \gamma$, by the transitive property, $b \cdot \gamma^* > b \cdot \widehat{\gamma}$.

4. $\Delta = \widehat{D}_\alpha \cup (A \oplus \{\beta\})$

   This breaks down into two cases: $\alpha = \widehat{\alpha}$ and $\alpha \neq \widehat{\alpha}$. If $\alpha = \widehat{\alpha}$ then $\widehat{\gamma} \in \widehat{D}_\alpha$, which means $\widehat{\gamma} \in \Delta$ and we revert to the argument of the previous case where $b \cdot \gamma > b \cdot \widehat{\gamma}$ implied $b \cdot \gamma^* > b \cdot \widehat{\gamma}$.

   When $\alpha \neq \widehat{\alpha}$ we compare γ against $A \oplus \{\beta\}$ and, since the region $R(\gamma, \Delta)$ is non-empty, we know that, as in the second case, $\forall \overline{\alpha} \neq \alpha \in A$ that

$$b \cdot (\alpha + \beta) > b \cdot (\overline{\alpha} + \beta)$$

$$b \cdot \alpha > b \cdot \overline{\alpha} \ .$$

   This implies that $b \cdot (\alpha + \widehat{\beta}) > b \cdot (\widehat{\alpha} + \widehat{\beta})$ and since $b \cdot \gamma^* \geq b \cdot (\alpha + \widehat{\beta})$, by transitivity we get $b \cdot \gamma^* > b \cdot \widehat{\gamma}$.

5. $\Delta = (\{\alpha\} \oplus B) \cup \widehat{D}_\beta$.

   This case follows from the previous case by symmetry.

■

Since each pass through the loop removed a vector from $\widetilde{D}$, the algorithm will terminate. Each vector removed from $\widetilde{D}$ is either discarded or added to $\widehat{D}$ and Theorems 3.3.1 and 3.3.2 show that only useful vectors are added to $\widehat{D}$ and all vectors discarded are useless; therefore $\widehat{D} = D$.

We note that although choosing $\Delta = \overline{D}$ or $\Delta = (\{\alpha\} \oplus B) \cup (A \oplus \{\beta\})$ results in a sound algorithm, there is not much need to ever choose these when the other choices are available. We know that $\widehat{D} \subseteq \overline{D}$, so choosing $\widehat{D}$ would always be better than choosing $\overline{D}$. Likewise, the last two sets both satisfy

$$\widehat{D}_\alpha \cup (A \oplus \{\beta\}) \subseteq (\{\alpha\} \oplus B) \cup (A \oplus \{\beta\})$$
$$(\{\alpha\} \oplus B) \cup \widehat{D}_\beta \subseteq (\{\alpha\} \oplus B) \cup (A \oplus \{\beta\}) \;\;,$$

making either a more preferable choice. However, without either of these last two choices, there are situations where choosing $(\{\alpha\} \oplus B) \cup (A \oplus \{\beta\})$ would be preferable to choosing $\widehat{D}$; specifically when $|\widehat{D}| > |A| + |B|$. This is important since one might not want to add the extra bookkeeping complexity required for implementing the most general form of the generalized cross-sum.

3.4 Other Exact Algorithms

In Section 3.3, we covered the previous enumeration-based algorithms so, in this section, we will focus on the algorithms more directly related to the witness algorithm, which search over information-space regions. We discuss four algorithms; two developed by Sondik [117] and two developed by Cheng [26]. Although the witness and incremental pruning algorithms were advances over the previous exact algorithms, they owe a great deal to the previous algorithmic approaches. More detailed discussion of these algorithms and their drawbacks can be found in other work [22, 74].

3.4.1 Sondik's Two-Pass

The witness, IP and GIP algorithms compute the value function one action at a time and then merge the resulting sets. This approach allows them to use techniques which would not be directly applicable if they tried to construct Γ_n directly. Although the witness algorithm was the first to use the single action value function approach by design, Sondik [117] uses this idea to motivate the design of the one-pass algorithm. The first pass sweeps through the information space to construct Γ_n^a and the second pass is required to merge these sets. Sondik refers to this as the one-pass algorithm applied to the single action problem, but since the one-pass algorithm requires a slightly different approach, we use the name *two-pass* when referring to this algorithm.

Section 3.4.2 presents the one-pass algorithm, but very little attention has been paid to the two-pass algorithm. Section 4.9 presents some empirical results concerning this algorithm, and here we present the algorithm both for its own merits and as the first step in explaining the one-pass algorithm.

Recall from Section 3.3.3 concerning the computation of the cross-sum of two sets, that we could determine if a vector $\alpha + \beta$ would be useful by defining the region $R(\alpha, A) \bigcap R(\beta, B)$ and checking to see if it was empty. From Equation 3.8 we see that all the vectors we need to consider in constructing Γ_n^a are simply elements of a cross-sum over all observations. Simply extending the argument we see that to determine if the vector

$$\begin{aligned} \gamma_n^a &= \sum_z \gamma_n^{a,z} \\ &= \gamma_n^{a,0} + \gamma_n^{a,1} + \gamma_n^{a,2} + \dots \end{aligned}$$

is useful, it is enough to check if the region

$$\bigcap_z R(\gamma_n^{a,z}, \Gamma_n^{a,z}) = R(\gamma_n^{a,0}, \Gamma_n^{a,0}) \bigcap R(\gamma_n^{a,1}, \Gamma_n^{a,1}) \bigcap R(\gamma_n^{a,2}, \Gamma_n^{a,2}) \bigcap \dots$$

is empty or not. Even more useful is the fact that

$$\bigcap_z R(\gamma_n^{a,z}, \Gamma_n^{a,z}) = R(\gamma_n^a, \Gamma_n^a) \quad ,$$

which simply defines the actual region for a given vector in Γ_n^a. This relationship is shown in Figure 3.12.

Sondik uses this fact to search the information space by defining a region for a vector, and then finding all the adjacent regions. Table 3.10 shows a routine for computing Γ_n^a using the two-pass algorithm. The two main sets of interest are: $\widehat{\Gamma}$ which holds the vectors we have found and whose regions we have already explored; and Υ which holds vectors which we have found, but whose regions we have not yet explored. Note that implementing this

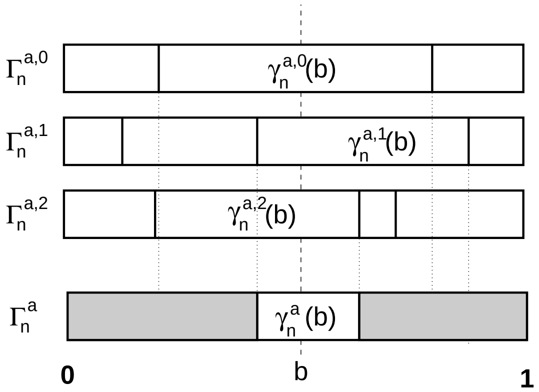


Figure 3.12: Defining the constraints on a region for Sondik's two-pass algorithm where $\gamma_n^a(b) = \gamma_n^{a,0}(b) + \gamma_n^{a,1}(b) + \gamma_n^{a,2}(b)$.

algorithm will require maintaining the information about how the individual γ_n^a vectors were constructed; i.e., which $\gamma_n^{a,z}$ vectors were used.

The outer loop of the algorithm is over the vectors whose regions we have not yet explored, and the inner loop is responsible for exploring the region. For each region, it looks at all the potential region borders; each *neighbor* vector (see Section 3.2.1 and Appendix E) may form one of the borders. For each neighbor it finds whether it forms a border of the region, and if so, adds it to Υ for later exploration.

The two-pass algorithm explores a region using an LP set up by the `setUpTwoPassLP` routine. The LP here has the variables $b(s)$, one for each state. The objective function is $b \cdot (\nu - \gamma_n^a)$ with the constraints $b \in R$ and $b \in \Pi(\mathcal{S})$. If a neighbor vector defines a border, then this LP should be feasible with solution points all along the border.

The main problem with the two-pass routine as shown is that there is no guarantee that it will return a parsimonious set. Vectors, which we termed

```
twoPass(Γ_{n-1}, a)
    Γ̂ := ∅
    b := any information state
    Υ := {γ_n^a(b)}
    do
        γ_n^a := removeElement(Υ)
        Γ̂ := Γ̂ ∪ {γ_n^a}
        R := ⋂_z R(γ_n^{a,z}, Γ_n^{a,z})
        for each ν ∈ 𝒩(γ_n^a)
            L := setUpTwoPassLP(γ_n^a, ν, R)
            if feasibleLP(L) and ν ∉ Γ̂
                then Υ := Υ ∪ {ν}
        end for each ν
    until Υ = ∅
    return Γ̂
end twoPass
```

Table 3.10: The two-pass algorithm for constructing Γ_n^a.

imposters (see Page 60), could be present in the final Γ_n^a set. In fact, it is necessary to include these imposter vectors, since not all adjacent regions are formed by neighbors as the counter-example of Appendix E shows. For the cases where adjacent regions to $R(\gamma, \Gamma_n^a)$ are not formed by neighbors, there are neighbors of γ which will be imposters. These imposters will have their own neighbors, different from γ, which will eventually lead to the adjacent region's vector.

It may be possible to avert this problem by moving just beyond the border of a region. This would ensure that we actually get an information point that lies in the interior of the adjacent region. However, there is the potential for missing regions if we move too much beyond the region's border. We have not explored any sophisticated methods for how this might be accomplished, so we are left with the unsatisfactory result of the `twoPass` routine possibly returning a non-parsimonious set.

As we will see in Section 4.1.4 that the witness algorithm has a worst case running time that is polynomial in the model size and the sizes of Γ_{n-1} and Γ_n^a. If not for the imposter vector problem, we could say the same for thing for the two-pass algorithm; in theory, there could be an exponential number of imposters even when the size of Γ_n^a was relatively small.

3.4.2 Sondik's One-pass

The one-pass algorithm[5] is based on the same ideas as the two-pass algorithm, but constructs Γ_n directly, thus foregoing the second pass required to merge the Γ_n^a sets. In the two-pass algorithm the region $\bigcap_z R(\gamma_n^{a,z}, \Gamma_n^{a,z})$ is sufficient to describe the actual region $R(\gamma_n^a, \Gamma_n^a)$ of a vector. This region is useful in the one-pass algorithm, however, it is not sufficient, since it must consider the effects of the other actions.

We do know that $R(\gamma_n^a, \Gamma_n) \subseteq R(\gamma_n^a, \Gamma_n^a)$ and so this region constraint becomes the starting point for defining the regions that the one-pass algorithm imposes.

When we are finding the vector for a point b, we must construct a vector for each action and see which is maximal. Let $\gamma^{a^*}(b)$ be the maximal vector with a^* being the best action. Restricting our attention to the region $R(\gamma_n^{a^*}(b), \Gamma_n^{a^*})$, if the $\gamma^a(b)$ for $a \neq a^*$ are always as shown in Figure 3.13,

[5]The one-pass algorithm is presented in both Sondik's thesis [117] and a journal article [116]. However, as has been discovered independently by many researchers [89, 76], the constraint set defined in the journal article is inadequate for finding the optimal solution. The details are also described by Cassandra [22].

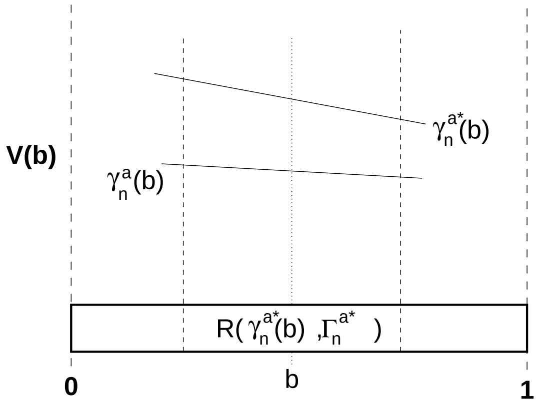


Figure 3.13: The case where the region constraints are adequate for defining the region.

then we are assured that $R(\gamma_n^{a^*}(b), \Gamma_n) = R(\gamma_n^{a^*}(b), \Gamma_n^{a^*})$, since there is no other action which will have a vector giving a higher value over the entire region. However, there are two ways that this region can be larger than the actual region $R(\gamma^{a^*}(b), \Gamma_n)$, i.e., $R(\gamma^{a^*}(b), \Gamma_n) \subset R(\gamma_n^{a^*}(b), \Gamma_n^{a^*})$, which means that the one-pass algorithm must further restrict the region. This restriction is necessary since too large of a region could completely contain some other regions, resulting in missing these if it only looks at adjacent regions.

The first set of extra constraints are put on the region to handle the case shown in Figure 3.14. Here, we find that a different one of the $\gamma^a(b)$ vectors has become maximal over the region $R(\gamma_n^{a^*}, \Gamma_n^{a^*})$. To ensure that this does not happen we add the constraint

$$R(\gamma_n^{a^*}, \{\gamma_n^a | \forall a\}) \ ,$$

which merely ensures that a^* stays optimal.

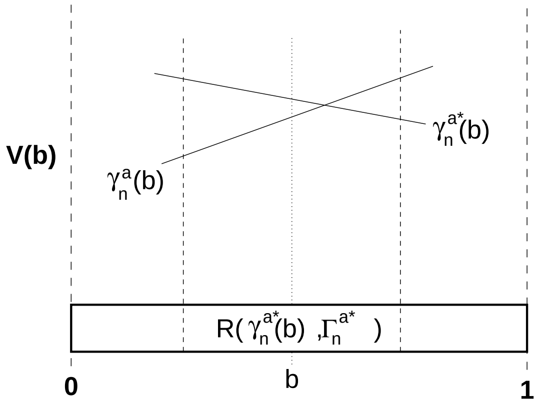


Figure 3.14: The first case where we must restrict the region.

Even with this added constraint, the region defined can still be a strict superset of $R(\gamma^{a^*}(b), \Gamma_n)$. Figure 3.15 shows the most troublesome case that the one-pass algorithm has to handle. Here we see a point b' which has the same vector as b for a^*; i.e., $\gamma^{a^*}(b) = \gamma^{a^*}(b')$. However, for another action a we have $\gamma^a(b) \neq \gamma^a(b')$. As Figure 3.15 shows, this other vector may yield higher values than $\gamma^{a^*}(b)$ at b'. For this reason, it is necessary to add the region restrictions $R(\gamma^a(b), \Gamma_n^a)$ for all actions. In the original Smallwood and Sondik paper [116] description of the algorithm, this restriction was inadvertently omitted as was also discovered by a number of other researchers [76, 89].

This makes the final set of constraints

$$\bigcap_{a,z} R(\gamma_n^{a,z}, \Gamma_n^{a,z}) \bigcap R(\gamma_n^{a^*}, \{\gamma_n^a | \forall a\}) \quad .$$

The unfortunate aspect of adding this last set of constraints is that we could now be defining a subset of the true region as shown in Figure 3.16. Here, the vector for action a has changed over the $R(\gamma^{a^*}(b), \Gamma_n^{a^*})$, but the vector

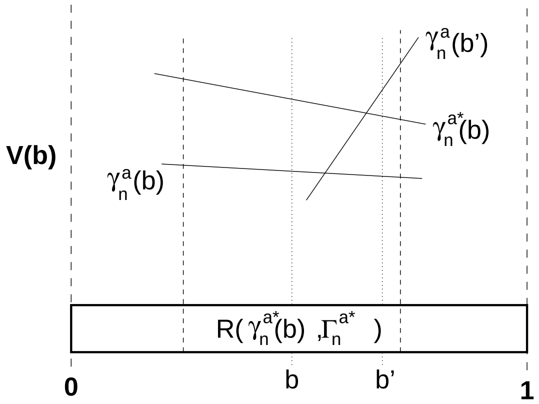


Figure 3.15: The case where we must restrict the region even further.

that becomes maximal for action a is still not better than γ^{a^*} anywhere within this region and the regions defined by the one-pass algorithm could be too conservative.

The fact that these regions can be smaller than the true regions adds some complications to the actual algorithm. It is no longer the case that the border of a region is a border with another actual region in Γ_n. Figure 3.16 shows that we could find a border of two regions where the same vector is maximal over both regions. Therefore, it is no longer enough to keep track of vectors which have and have not had their regions checked, since we might have to check the same vector over a few different regions. This will require us not only to keep track of which vectors we have searched, but the context in which the regions were searched (i.e., what were the other γ^a vectors for the region.) An alternative implementation approach is to maintain a set of information points and define a loop over this set, however this approach is also complicated by the fact that the points generally lie on the region

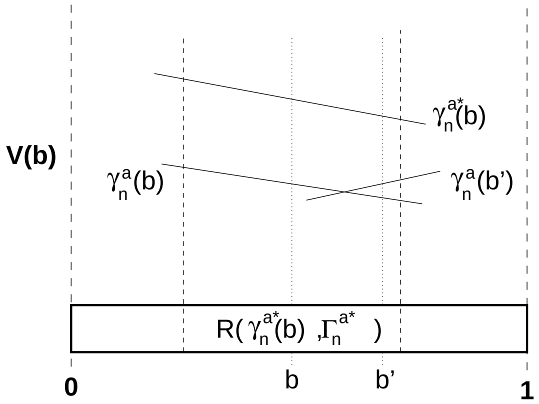


Figure 3.16: The case where the further restriction is unnecessary.

borders.

Despite some of the shortcomings mentioned, Sondik's *one-pass* algorithm was the first exact algorithm and provided the inspiration for all of the subsequent algorithms.

3.4.3 Cheng's Relaxed Region

In his thesis [26], Cheng presents a variation of the one-pass algorithm which avoids the overly restrictive regions. Cheng refers to this as the *relaxed region* algorithm, since he simply removes some of the one-pass algorithms constraints. The constraints imposed by the relaxed region algorithm are

$$\bigcap_z R(\gamma_n^{a,z}, \Gamma_n^{a,z}) \bigcap R(\gamma_n^{a^*}, \{\gamma_n^a | \forall a\}) \ \ ,$$

which just omits the last region restriction discussed in the section on the one-pass algorithm.

As mentioned in the one-pass discussion, allowing the regions to get too large can sometimes mask regions that lie in the interior. To combat this, Cheng has to do more than find a single point along a region boundary, which is all that was required for the one-pass algorithm. To ensure that the relaxed region algorithm doesn't miss any interior regions, it must check every vertex of the defined region. Cheng uses a vertex enumeration algorithm [80] on the regions defined and checks each vertex to see if a previously undiscovered vector is maximal there.

The computational complexity of doing this enumeration is exponential in either the dimension of the state space or the number of binding constraints of the region, whichever is smaller. In the relaxed region algorithm, the number of binding constraints is related to the number of observations and the size of Γ_{n-1}. This exponentiality is a direct result of the complexity of the number of vertices in a convex polytope [81]. Although this is a worst case complexity result, it does seem to be the limiting factor in practice and rarely can the relaxed region algorithm solve a problem with more than 4 or 5 states.

3.4.4 Cheng's Linear Support

For Cheng, the relaxed region algorithm was primarily the gateway to a more sophisticated algorithm, and one which was the inspiration behind the witness algorithm; the *linear support* algorithm. From the ideas of the relaxed region algorithm, we see that it is possible to define overly optimistic regions; i.e., a vector's region in the approximation is larger than it will be with respect to the final Γ_n set.

In all the previous approaches discussed in this section, a vector is compared against other "potential" vectors that could be constructed from Γ_{n-1}.

With the linear support algorithm, Cheng introduced the idea of gradually building up Γ_n and comparing any new found vector to the current vectors already discovered. If we let $\widehat{\Gamma} \subseteq \Gamma_n$ be the set of vectors as we build Γ_n, then for a given b and its vector $\gamma^{a^*}(b)$, we can define the region $R(\gamma^{a^*}(b), \widehat{\Gamma})$. Because $\widehat{\Gamma}$ is a subset of Γ_n, we know that this region must be a superset of the actual region.

Unfortunately, the linear support algorithm still requires the same method of checking each vertex to ensure no regions are missed. As discussed for the relaxed region, this is not practical for problems with more than a few states. However, the idea of comparing a vector to an evolving subset of the true value function representation is very useful and appears in the PRUNE, the witness and the GIP algorithms.

3.5 Conclusions

Although there were many preliminary results and basic theory development for POMDPs prior to 1970 [38, 2, 120, 1, 40, 3, 106, 101, 36, 109, 114], the first exact algorithm for solving the general POMDP problem was developed by Sondik [117, 116] under the name of the "one-pass" algorithm. Although there was subsequent research [129, 87, 39, 26, 132], the computational complexity of the problems themselves, the intricacies of the algorithm and the lack of computing power of the day all combined and seemed to limit the amount of exploration and number of researchers involved in further developing the theory and algorithms for POMDPs.

More recently there has been a resurgence of interest in the POMDP model and whereas the majority of the previous work was in developing the essential theory, this recent interest has been spurred on by trying to apply these theories to some more realistic problems and focusing on the algorithmic issues. The witness, incremental pruning and generalized incremental pruning algorithms are results of this more recent algorithmic perspective.

The majority of the effort into exact or near exact algorithms for solving POMDPs has been based upon value iteration. However, Sondik [118] presented a policy iteration algorithm for POMDPs, though the implementation of this algorithm presents many challenges and has not yet been shown to be useful for more realistic sized problems. More recently Hansen [45] has revisited the policy iteration approach, resulting in a simpler and more effective algorithm. Although the dynamic programming updates discussed in this chapter are the basis of value iteration solutions, Hansen's policy iteration algorithm uses these DP updates in his policy improvement phase. Thus, more effective exact (or approximate) DP updates can translate directly into more effective policy iteration algorithms.

Chapter 4

Analysis of Exact Algorithms

We begin this chapter with a brief review of some existing complexity results concerning the solution of POMDPs in general and some specific results concerning POMDP algorithms. We will see that four algorithms, two-pass, witness, IP and GIP, fall into the same general complexity class. We then undertake a detailed analysis of these four algorithms. Since we will also see that the general single step DP problem of computing $V_n(\cdot)$ from $V_{n-1}(\cdot)$ is hard, we focus on the complexity of computing $V_n^a(\cdot)$ from $V_{n-1}(\cdot)$. All of the algorithms considered here are methods for constructing Γ_n^a from Γ_{n-1} and have essentially the same asymptotic worse case complexity when viewed simply from the problem size. However, the more detailed analysis in this chapter shows that some algorithms are more efficient than others.

We approach the analysis from the number and sizes of the linear programs that need to be solved. The dependence upon linear programming is clear from the algorithm specifications, but we have also verified this on a range of problems and found that 90% to 95% of the computation time is spent in the linear programming routines[1]. Since detailed analysis of the LP algorithms and their interaction with the specific LPs set up in trying to solve a POMDP is quite complex, we simplify the analysis to incorporate only the number of LPs and the total number of constraints needed for each algorithm. We will see towards the end of this chapter that the empirical evidence supports the validity of this analysis.

Our analysis will proceed from the simplest routines upward to the full algorithms. We first derive formulas for the numbers and sizes of the LPs for each algorithm and then proceed to compare them. We ignore the number of variables in the LPs, since this corresponds to the number of states in the POMDP and is constant for a given problem across algorithms. For all of the LPs, there is an additional constraint that confines the solutions to be probability distributions, since the variable is the information state vector. We ignore the non-negativity constraints, since non-negativity is a natural constraint for LP solvers. However, we will include the *simplex constraint*, $\sum_s b(s) = 1$, since this explicitly needs to be stated in the LP and since its omission would unfairly favor algorithms that required a lot of small LPs over algorithms that do fewer, but larger LPs.

We preface the remaining sections of this chapter by noting that due to the computational complexity of solving a POMDP, exact algorithms by themselves are not much use. However, this does not discredit the use or need to look at these algorithms. Exact algorithms can often directly lead to approximations or provide insight into the solutions which can be exploited

[1]We used an efficient commercial LP solver to ensure this was not an artifact of the implementation.

in other algorithms. Furthermore, some of the subproblems solved in the course of the exact algorithms can be used in approximation schemes [108, 132].

4.1 Computational Complexity of POMDPs

Before undertaking our more detailed analysis, we present the existing complexity results for solving POMDPs in general, solving POMDPs via value iteration and the relative complexity of the existing algorithms.

4.1.1 Background

Before discussing the existing complexity results on MDPs, we will very briefly introduce the basic ideas we will need from computational complexity theory. This simplified view will provide enough background to understand the main complexity results as they pertain to MDPs. Much more comprehensive treatments of computational complexity theory are found in many textbooks [66, 47, 124].

Theoretical computer science has been useful in classifying the problems according to their computational hardness. The class P represents the easy or tractable problems which can be solved in a polynomial amount of time. The class NP is a broader class, incorporating P, and includes problems that are typically accepted to be hard or intractable. Problems that are NP-complete are the hardest problems in the class NP and are problems which are assumed to take an exponential amount of time. This idea of completeness extends to any of the problem classes; it is the set of problems that are the hardest to solve in the class. Although P $\subseteq$ NP, it is not known whether P is a proper subset or not, though the general belief in the theoretical computer science community is P $\neq$ NP.

There is even a broader class, PSPACE, which are the problems which can be solved in a polynomial amount of space. The class P is included in this class, P $\subseteq$ PSPACE, since any algorithm that runs in polynomial time, can only consume a polynomial amount of resources. Slightly less intuitive, is the fact that NP $\subseteq$ PSPACE. To understand this requires a more formal definition of NP, which we will not give here. Even for this wider class, whether equality holds in either case is an open theoretical question. Even in the unlikely case that P = NP, this will not necessary imply that P = PSPACE.

Therefore, we adopt the commonly accepted viewpoint: problems in P are ones that are considered solvable; NP-complete problems are considered to require an exponential amount of time; and PSPACE-complete problems are considered to be even harder than NP-complete problems.

4.1.2 Complexity of Exact Algorithms

COMDPs

We can solve the infinite horizon COMDP problem by casting it a (reasonably sized) LP [78, 63] and solving. Since the complexity of linear programming is a well known P-complete problem, we can see that solving an infinite horizon COMDP is easy from a complexity perspective. More generally, for both the infinite and finite horizon COMDP problem, computing the optimal policy is a P-complete problem as shown by Papadimitriou and Tsitsiklis [95].

POMDPs

Unfortunately, the nice computational aspects of the problem disappear with the introduction of partial observability. As was also shown by Papadimitriou and Tsitsiklis [95], finding the optimal policy for even a simplified finite horizon POMDP is PSPACE-complete. In their work, they consider a POMDP where $|\mathcal{Z}| < |\mathcal{S}|$, the observations are deterministic and where the horizon length is $T < |\mathcal{S}|$. Clearly, the more general POMDPs discussed here can be no easier to solve for the finite horizon. Even under many other forms of restrictions in model size, horizon length and type of policy, exact POMDP solutions are hard [20, 92].

Also pointed out by Papadimitriou and Tsitsiklis is the unlikeliness of even being able to represent a finite horizon policy with a polynomial-sized data structure. Thus, even if we were willing to expend exponential (or more) time to compute the optimal policy, an optimal controller based upon the resulting policy would be impractical to implement.

The situation appears to be even worse for the infinite horizon case. Papadimitriou and Tsitsiklis speculated that no finite algorithm will be able to exactly compute the infinite horizon policy and others have suggested that the problem is undecidable.

Single DP Step

With the result that finding optimal finite horizon solutions for POMDPs is hard, even for small horizons, we immediately see that just one DP stage (i.e., horizon equals 1) is also hard. This results from the Γ_n set of vectors produced from Γ_{n-1} possibly having size $|\mathcal{A}||\Gamma_{n-1}|^{|\mathcal{Z}|}$, which is exponential in the number of observations. This corresponds to every possible vector that could be constructed have some non-empty useful region. The conclusion from this is that we cannot have a polynomial-time algorithm for computing Γ_n from Γ_{n-1}.

The question that arises first is whether or not there exist POMDP problems can really exhibit this worst case behavior. Littman [72] shows just such a class of POMDPs, which leads to the question of whether or not useful POMDPs (i..e, ones that need to be solved) exhibit this worst case behavior. While this question remains unanswered, predominantly due to the lack of many such models, we then are led to wondering how hard it might be to solve POMDP problems which do not exhibit this worst case behavior; i.e., $|\Gamma_n|$ is not exponential in $|\mathcal{Z}|$.

This leads to considering a class of problems that are *polynomial output-bounded*, which simply says that the size of the answer is restricted to be polynomial in the size of the input. Littman [72] has also shown that this too is a hard problem.

4.1.3 Complexity of Approximations

We now briefly divert ourselves from exact solutions to discuss some complexity results for approximate solutions. Considering the computational challenges presented by POMDPs, Chapters 5 and 6 will explore developing approximate solutions. Unfortunately, little can be said about theoretical guarantees of the quality of their solutions. In the best scenario, we would be able to develop algorithms with guaranteed performance criteria in relation to the optimal answer. There is an entire sub-field of computer science which addresses approximation algorithms from this perspective. Some problems which are hard to solve exactly have a corresponding easy approximation algorithm which can give a guarantee on the closeness of the approximation.

Within this sub-field, researchers have also come across classes of problems which are just as hard to approximate as they are to compute exactly. Unfortunately, POMDPs are one such class of problems, even for the finite horizon case. Condon and Feigenbaum [30] have some results about the hardness of approximating some PSPACE-hard problems, which translate nearly directly into the hardness of approximating finite horizon POMDP problems. A thorough complexity analysis of POMDPs and their hardness of approximation results can be found in the work by Goldsmith, Mundhenk, Lusena and Allender [44, 91, 92]. In this work, they show complexity results for a broad range of POMDPs under various restrictions, none of which have tractable algorithms for their solution.

4.1.4 Complexity of POMDP Algorithms

Returning to the realm of exact algorithms, and having left off with the unpleasing result that performing even a single DP step is hard, we can attempt to narrow the problem further to see if there is any part of solving POMDPs which may be tractable.

Recall that the algorithms, witness, IP, GIP and two-pass, do not construct Γ_n directly, but instead consider the sub-problem of constructing the Γ_n^a sets and merging the results. The first way we could consider approaching the problem is to address the complexity of constructing Γ_n^a from Γ_n. Unfortunately, there is more bad news, since the sizes of the Γ_n^a can be exponential in the number of observations; i.e., as large as $|\Gamma_{n-1}|^{|\mathcal{Z}|}$. The fact that such worst case problems do exist is an immediate corollary of there being worst case problems for the full Γ_n construction.

As we did for the single DP step case, we could then restrict our attention to problems where the answer is not exponential in the size of the inputs, where here the inputs are $\mathcal{S}$, $\mathcal{A}$, $\mathcal{Z}$ and Γ_{n-1}. This class of POMDPs are referred to as *polynomial action output-bounded* and present the first restricted class of POMDP problems which are tractable. We will see as immediate results from the detailed analysis to follow that the witness, IP and GIP algorithms can all solve this class of problems in polynomial time in the worst case. We will also see that the two-pass algorithm has a best case complexity that allows the solutions of these problems in polynomial time, though the worst case is exponential.

Concerning the previously existing algorithms, the batch enumeration algorithm of Section 3.3.1 and the linear support algorithm of Section 3.4.4, we have discussed how these are exponential time algorithms for constructing Γ_n. This is not surprising, since any algorithm for constructing Γ_n must have a worst case exponential running time. However, even if we suitably adjusted these algorithms to solve polynomial action output-bounded algorithms, they still exhibit exponential worst case performance. The batch enumeration still needs to enumerate an exponential number of vectors and the linear support algorithm still needs to enumerate vertices of convex polytopes. We note that the suitable adjustment of Sondik's one-pass algorithm for attacking polynomial action output-bounded POMDPs is exactly the two-pass algorithm, whose computational characteristics we discuss below in detail.

4.2 The PRUNE Algorithm

The PRUNE routine from Table 3.4 in Section 3.1.1 is the fundemental building block for many of the algorithms we analyze, so we begin the analysis here. The PRUNE routine as presented, starts with a single set of vectors, $\overline{\Gamma}$, to be pruned and an initializes the set it constructs, $\widehat{\Gamma}$, to be the empty set. In truth, $\widehat{\Gamma}$ can be initialized with any number of *useful* vectors from the set Γ. This is discussed further in Section 4.7.1, but for now we assume it is initialized to the empty set.

For notational simplicity, we will let $|\overline{\Gamma}| = G$ and let $|\Gamma| = V$ be the size of the final parsimonious set. Thus we will express the complexity in terms of the initial and final sizes of the sets.

Total LPs

The total number of LPs required is equal to the number of vectors in Γ. Without any prior knowledge, we must check every vector to see if it is useful or not. Notationally this is

$$\text{PRUNE}_{\text{LP}}(G, V) = G \quad .$$

4.2.1 Total Constraints

There are at least three ways we could attempt to characterize the total number of constraints required: worst case, best case and average case. These cases arise because the actual sizes of the LPs that are solved depends upon the number of useless vectors and the order in which we process the vectors in the loop of the PRUNE algorithm.

Total Constraints - Worst case

The worst case for the PRUNE algorithm is when the sizes of the LPs get as large as possible as quickly as possible. Another way to see this is that the worst case is when every useless vector must be compared against every useful vector. Thus the worst case is when the first V vectors chosen are useful vectors. The remaining $G - V$ useless vectors will then be compared to all V vectors. The first LP has only the simplex constraint and as each useful vector is added, the number of constraints increases by one. When all useful vectors have been uncovered, all the subsequent LPs for the useless vectors will have a constraint for each of the V vectors plus the simplex constraint yielding

$$\begin{aligned} \text{PRUNE}_{\text{worst}}(G, V) &= \sum_{i=1}^{V} i + \sum_{i=1}^{G-V} (V+1) \\ &= \frac{V(V+1)}{2} + (G-V)(V+1) \\ &= (V+1)\left(G - \frac{1}{2}V\right) \quad . \end{aligned}$$

Note that it is very unlikely that the vectors would be processed in such an unfavorable manor.

Total Constraints - Best case

To get a better feel for how loose or tight the worst case is, we look at the absolute best case for the PRUNE algorithm. This case is just the opposite of the worst case and in some regards, even more unlikely to happen. In this case, the first LP contains only the simplex constraint and for any given point returned, it must lead to a useful vector. However, after this, if this one vector, by itself, dominates all of the useless vectors in G, then we can eliminate all the useless vectors using an LP with only 2 constraints. Processing the remaining useful vectors afterwards yields

$$\begin{aligned}
\text{PRUNE}_{\text{best}}(G,V) &= 1 + \sum_{i=1}^{G-V} 2 + \sum_{i=1}^{V-1}(i+1) \\
&= 1 + 2(G-V) + \frac{V(V-1)}{2} + V - 1 \\
&= 2G + \frac{1}{2}V(V-3) \ .
\end{aligned}$$

Total Constraints - Average case

As mentioned, both the best and worst case analysis for the PRUNE algorithm account for improbable orders of vector selection. In an attempt to get a measure closer to reality, we suppose that the vector selection proceeds as follows: at each iteration of the loop, we are equally likely to select any of the remaining vectors. Furthermore, we assume that if we select a useless vector, that it *does not* lead to an information state that would lead to finding a useful vector. Note that in general, we could select a useless vector, which compared to the current approximation looks useful and the information state at which it looks useful will serve to find the true useful vector. This is not completely realistic either, since finding a useful vector is more likely than we assume here, since we assume useful vectors are found *only* by selecting them directly for checking against the current approximation.

However, given this assumption we can define the average number of constraints with the recursion

$$C(i, g, v) = i + 1 + \frac{g - v}{g} C(i, g - 1, v) + \frac{v}{g} C(i + 1, g - 1, v - 1) \quad ,$$

where i is the number of useful vectors found so far, g the total number of vectors left to check and v the number of the remaining vectors in g which are useful. The base cases are

$$\begin{aligned} C(i, 0, 0) &= 0 \\ C(i, g, 0) &= i + 1 + C(i, g - 1, 0) \\ C(i, x, x) &= i + 1 + C(i + 1, x - 1, x - 1) \quad . \end{aligned}$$

This recursion has the closed form

$$C(i, g, v) = g \left(\frac{v}{2} + i + 1 \right) - \frac{v}{2} \quad ,$$

which can be verified inductively. This makes the average number of constraints

$$\mathrm{PRUNE}_{\mathrm{ave}}(G, V) = \frac{1}{2} V(G - 1) + G \quad .$$

Note that since finding useful vectors is more likely than we assume, the true average case is larger than this quantity. However, analyzing the true average case requires intimate knowledge of the shape of the value function and the other useless vectors. This would require sophisticated analysis dealing with probability distributions on PWLC value functions and vectors. The worst case provides an upper bound and this unrealistic average case provides a lower bound, which is more than sufficient for our purposes here.

4.3 Cross-sum Algorithms

In this section we look at three ways in which the parsimonious representation of the cross-sum of two sets can be computed. This will correspond to variations available in the IP and GIP algorithms. For all of this discussion, we will assume that the sets A and B are the two sets being cross-summed, where $|A| = N$ and $|B| = M$ and the final parsimonious representation to be computed is of size C. Without loss of generality, we will assume that $N \geq M$, making A the larger set. Additionally, we assume that the sets A and B are themselves parsimonious, which constrains the size of the resulting set, $N \leq C \leq NM$.

4.3.1 Normal Cross-sum

We define the *normal cross-sum* (NCS) to be the algorithm that simply enumerates all possible vectors in the cross-sum and then uses the PRUNE routine to reduce it to its parsimonious set. This is the exact approach used in the regular version of IP. As a result, the analysis for this is simply the analysis for the pruning algorithm. We have

$$\begin{aligned} \mathrm{NCS}_{\mathrm{LP}}(N, M, C) &= \mathrm{PRUNE}_{\mathrm{LP}}(NM, C) \\ &= NM \ , \end{aligned}$$

$$\begin{aligned} \mathrm{NCS}_{\mathrm{worst}}(N, M, C) &= \mathrm{PRUNE}_{\mathrm{worst}}(NM, C) \\ &= (C+1)\left(NM - \frac{1}{2}C\right) \ , \\ \mathrm{NCS}_{\mathrm{best}}(N, M, C) &= \mathrm{PRUNE}_{\mathrm{best}}(NM, C) \\ &= 2NM + \frac{1}{2}C(C-3) \ , \end{aligned}$$

and

$$\mathrm{NCS}_{\mathrm{ave}}(N, M, C) = \mathrm{PRUNE}_{\mathrm{ave}}(NM, C)$$
$$= \frac{1}{2}C(NM-1) + NM \quad .$$

4.3.2 Restricted Region Cross-sum

Recall from Section 3.3.3 that there were five different sets to choose from in the GIP algorithm. Since the GIP algorithm is essentially the IP algorithm with a more general cross-sum operation, we defined the cross-sum operations of the GIP algorithm as generalized cross-sums (GCS).

Consider the GCS algorithm where instead of allowing the freedom to choose from among the five sets shown on Page 88, we analyze the situation when we decide up-front which set we will use, and use that set for every iteration of the cross-sum. We define the *restricted region* (RR) cross-sum as the GCS algorithm where one of either

$$\Delta = \widehat{D}_\alpha \cup (A \oplus \{\beta\})$$

or

$$\Delta = (\{\alpha\} \oplus B) \cup \widehat{D}_\beta$$

is always chosen. Note that the variation of GCS which always chooses $\Delta = \widehat{D}$ is simply the NCS algorithm.

The analysis for both of the choices above is exactly the same; the only difference is in which set we choose to be the *restricting set* (see Page 90). For those two cases above, the former has A as the restrictor and the latter has B as the restrictor. We will see that deciding to make the larger or smaller of A and B the restrictor set *does* have an influence on the sizes of the LPs to be solved. Later, on Page 124, we discuss the effects of the set ordering in the RR cross-sum. For the analysis of RR we assume that we have chosen the set $\Delta = \widehat{D}_\alpha \cup (A \oplus \{\beta\})$. Since we need to be sensitive to which set is the restrictor, we define F to be the size of the restricting set A and let the other set, B, have size L and do not impose any restriction upon their relative sizes.

We will see that the RR algorithm is just an instance of GCS and as such requires just as many LPs and any other non-optimized variation of GCS. Thus the number of LPs required is FL as discussed below in Section 4.3.3,

We can simplify the analysis of the total number of constraints by viewing the RR algorithm as simply the PRUNE routine with an extra set of constraints for each LP. The RR algorithm is essentially attempting to prune the vectors in B, but instead of looking at the entire information space simplex, is restricted to the region $R(\alpha, A)$. This is then repeated for each $\alpha \in A$, hence our use of the term *restricting set* for the set A.

Using this insight, then for all cases we see that the total number of constraints for all cases is

$$\mathrm{RR}_*(F, L, C) = \sum_{i=1}^{F} [\mathrm{PRUNE}_*(L, C_i) + L(F-1)] \quad , \tag{4.1}$$

where F is the size of the restrictor set, L the size of the other set and C_i are the number vectors we find useful from B over the restricted region of the i^{th} vector from A. The asterisk in the subscript serves as a wild-card to be replaced with either `best`, `worst` or `ave`. Note that $\sum_i C_i = C$, $\forall i, C_i \geq 1$ and $C \geq \max(F, L)$. The $L(F-1)$ term in the summation arises from the fact that for each region we must do L LPs, and that we have an additional $F-1$ constraints to restrict the LP to only consider the region $R(\alpha, A)$.

Total Constraints - Worst case

We now plug in and show how to simplify the closed form for the worst case number of constraints for the RR algorithm.

$$\begin{aligned}
\mathrm{RR}_{\mathrm{worst}}(F,L,C) &= \sum_{i=1}^{F}\left[\mathrm{PRUNE}_{\mathrm{worst}}(L,C_i) + L(F-1)\right] \\
&= \sum_{i=1}^{F}\mathrm{PRUNE}_{\mathrm{worst}}(L,C_i) + F(F-1)L \\
&= \sum_{i=1}^{F}\left[(C_i+1)\left(L-\frac{1}{2}C_i\right)\right] + F(F-1)L \\
&= \sum_{i=1}^{F}\left[LC_i + L - \frac{1}{2}C_i^2 - \frac{1}{2}C_i\right] + F(F-1)L \\
&= L\sum_{i=1}^{F}C_i + FL - \sum_{i=1}^{F}\frac{1}{2}C_i^2 - \sum_{i=1}^{F}\frac{1}{2}C_i + F(F-1)L \ .
\end{aligned}$$

Recalling that $\sum_i C_i = C$ we get

$$\mathrm{RR}_{\mathrm{worst}}(F,L,C) = LC + FL - \sum_{i=1}^{F}\frac{1}{2}C_i^2 - \frac{1}{2}C + F(F-1)L \ .$$

Although we do not necessarily know the values of the individual C_i terms, since this is a worst case analysis we can replace that term $\sum_{i=1}^{F} C_i^2$ with anything that is guaranteed to be smaller, since by too little we preserve the worst case criteria.

If we have the general problem of minimizing $\sum_{i=1}^{n} x_i^2$ subject to the constraint $y = \sum_{i=1}^{n} x_i$, then the minimum is achieved[2] when $\forall i, x_i = y/n$.

[2]Thanks to Hagit Shatkay for verifying this.

For our case this means that

$$\sum_{i=1}^{F} \left(\frac{C}{F}\right)^2 \leq \sum_{i=1}^{F} C_i^2 \quad .$$

Using this simplification, we get

$$\begin{aligned}
\mathrm{RR}_{\mathrm{worst}}(F, L, C) &\leq LC + FL - \frac{1}{2}\sum_{i=1}^{F}\left(\frac{C}{F}\right)^2 - \frac{1}{2}C + F(F-1)L \\
&\leq LC + FL - \frac{C^2}{2F} - \frac{1}{2}C + F(F-1)L \\
&\leq F^2L + LC - \frac{C}{2}\left[\frac{C}{F} - 1\right] \quad . \qquad (4.2)
\end{aligned}$$

Note that this puts an upper bound on the worst-case complexity of the fully general GCS algorithm, since the RR algorithm does not have the freedom to choose a possibly smaller set to compare vectors against.

Total Constraints - Best case

We omit the full derivation of this case, but the simplifications made use of the same arguments as in the worst case scenario. Here we use the same simplification of replacing $\sum_{i=1}^{F} C_i^2$ with $\sum_{i=1}^{F}(C/F)^2$, which is valid because this quantity is added here and not subtracted.

$$\begin{aligned}
\mathrm{RR}_{\mathrm{best}}(F, L, C) &= \sum_{i=1}^{F}\left[\mathrm{PRUNE}_{\mathrm{best}}(L, C_i) + L(F-1)\right] \\
&= F^2L + FL + \frac{C}{2}\left[\frac{C}{F} - 3\right] \quad .
\end{aligned}$$

It is not obvious that is quantity is smaller than the worst case, but the relationships of the sizes of F and L to C and restriction on the maximum size, $C \leq LF$, does indeed lead to a smaller quantity.

Total Constraints - Average case

We will not need the derivation of the average case here, though it can be found by substituting directly from Equation 4.1, Note that the substitution of $C_i = C/F$ is still reasonable, since if we are looking to find an average case, then assuming that the average case equally distributes the C vectors among the F regions would require exactly this substitution.

Set ordering in RR

From Equation 4.2 we see that the worst case number of constraints is greatly effected by the ordering of the sets in the cross-sum. The dominating term is F^2L (unless $C > F^2$) which hints to select the smallest set as the restricting set. Even in the worst case where C is as large as possible, $C = FL$, the total constraints becomes

$$F^2L + FL^2 - \frac{1}{2}FL(L-3) \ ,$$

and we would still prefer to have the smaller set as the restrictor.

We empirically ran millions of examples for varying values of F and L and using the smaller set as the restricting set never results in more constraints. Although somewhat tedious, we believe that it can be proven that the worst case for choosing the smallest set as the restrictor is always better than the worst case of choosing the larger set. Given that this is just worst case, and in actuality choosing the larger could be better, we have not expened the effort on the proof, however there are definitely cases where the best case for choosing the larger set is better than the worst case for choosing the smaller set.

Therefore, in the remainder of the discussion and for our empirical evaluations, the restricted region cross-sum will imply that we choose the smallest set as the restricting set. Using $F = M$, $L = N$ and $N \geq M$ we get

$$\begin{aligned} \mathrm{RR}_{\mathrm{worst}}(M, N, C) &= M^2N + NC - \frac{C}{2}\left[\frac{C}{M} - 1\right] \\ \mathrm{RR}_{\mathrm{best}}(M, N, C) &= M^2N + MN + \frac{C}{2}\left[\frac{C}{M} - 3\right] \quad . \end{aligned}$$

4.3.3 Generalized Cross-sum

Total LPs

Regardless of how the choices for the Δ are made, all of the variations discussed require eventually checking all NM vectors, ignoring the possible use of the ideas in Section 4.7.1. Therefore, for any version of the generalized cross-sum, the total number of LPs is the same:

$$\begin{aligned} \mathrm{GCS}_{\mathrm{LP}}(N, M, C) &= \mathrm{PRUNE}_{\mathrm{LP}}(NM, C) \\ &= NM \quad , \end{aligned}$$

which is no different from the NCS cross-sum's total LP requirements, since it is an instance of GCS.

Total Constraints

Unfortunately, the most general, and most efficient, version of GCS is hard to directly analyze for the total number of constraints. The GCS algorithm provides the freedom to choose whichever of the five sets are smaller. Doing this in general requires knowing the exact order vectors are chosen and the results of all previous iterations, which are dependent upon the exact structure of the previous and resulting value functions as well as the implementation specifics. Without this knowledge, we would need to put some probability estimates on these quantities which would be a impossible without knowing the detailed structure of the problem instance. However, we have examined two variations of GCS, NCS and RR, which both use a single set for every iteration. Since the GCS algorithm has the freedom to choose between these sets (and others) the analysis for those two variations must provide an upper bound on the complexity of the GCS algorithm. Thus, we can characterize the GCS algorithm with

$$\mathrm{GCS}_*(N,M,C) = \min\left(\mathrm{NCS}_*(N,M,C), \mathrm{RR}_*(M,N,C)\right) \quad .$$

We will see in Section 4.8.1 that for the worst case total constraints the $\mathrm{RR}_{\mathrm{worst}}(M,N,C)$ is almost always smaller than $\mathrm{NCS}_{\mathrm{worst}}(M,N,C)$. Keep in mind that this is merely an upper bound on the most general form of GCS.

4.4 Incremental Pruning

With the analysis of the various cross-sum algorithms in hand, we can now embark upon analyzing the incremental pruning algorithms, since they are simply repeated applications of the cross-sum operations. Note that our references to the cross-sum algorithms here are for computing the parsimonious representation of the cross-sum.

4.4.1 Set Ordering in IP

The first issue that arises in the context of the incremental pruning algorithms is the set ordering. Recall from Equation 3.10 that the IP algorithm is given by interleaving PRUNE operations in the expression

$$\Gamma_n^{a,0} \oplus \Gamma_n^{a,1} \oplus \Gamma_n^{a,2} \oplus \ldots \oplus \Gamma_n^{a,|\mathcal{Z}|-1} \quad .$$

However, since the cross-sum operation is both associative and commutative, we are free to order and group the individual $\Gamma_n^{a,z}$ sets in any way desired. Note that this is also an issue with the GIP variations, except that the PRUNE operation is replaced with the more general GCS algorithm of Table 3.9.

We have seen that the ordering of the sets could have an impact on the number and sizes of the LPs that need to be solved, especially in the restricted region cross-sum variation. Additionally, this problem seems to have the same basic flavor as the *matrix chain multiplication problem* [31], which is a problem where the proper parenthesization can have a profound effect on the computational requirements. As in matrix chain multiplication, we could postulate using dynamic programming to decide the best parenthesization, where we use the formulas for the various cross-sum operations to compute the number and sizes of the LPs. However, unlike the matrix multiplication operator, the cross-sum operation is commutative, which means that we have to choose an ordering as well as a parenthesization. Another difference is that in matrix multiplication, we know the size of the resulting matrix, where in general we will not know the resulting size of the parsimonious representation.

Recall from Table 3.8 that the IP algorithms maintain a set, Ψ, of sets of vectors and at any given point in time selects two sets, computes the parsimonious representation of the cross-sum and puts the result back into Ψ. Among the more obvious choices for deciding which two sets to choose are

- selecting the two smallest sets (IP-SS),
- selecting the two largest sets (IP-LL) or
- selecting the largest and the smallest sets (IP-SL).

Somewhat surprisingly, even though we will not know the sizes of the resulting cross-sums, we can completely characterize the parenthesization for the IP-LL and IP-SL cases. To show this, assume that we have a set of sets of vectors $\Psi = \{\Gamma_i | 1 \leq i \leq n\}$ where $\forall j > i, |G_i| < |G_j|$, i.e., the sets are indexed according to size. Then we have

$$(\Gamma_1 \oplus (\ldots \oplus (\Gamma_{n-2} \oplus (\Gamma_{n-1} \oplus \Gamma_n)) \ldots)) \quad (\text{IP-LL})$$

and

$$((\ldots(((\Gamma_n \oplus \Gamma_1) \oplus \Gamma_2) \oplus \Gamma_3) \oplus \ldots) \oplus \Gamma_{n-1}) \quad (\text{IP-SL}) \ .$$

The reason these orderings are completely determined is the fact that

$$|\text{PRUNE}(\Gamma_i \oplus \Gamma_j)| \geq \max(|\Gamma_i|, |\Gamma_j|) \ .$$

Whenever the largest set of the group is selected, we are assured that the result will be as large as the chosen set, which means that we will select the resulting set on the next iteration.

For the IP-SS case it is impossible to completely characterize the set selection ordering. Although the first cross-sum is guaranteed to select Γ_1 and Γ_2, subsequent selections depend upon sizes of the resulting sets. If $|\text{PRUNE}(\Gamma_1 \oplus \Gamma_2)| < |\Gamma_4|$, then the result of $\Gamma_1 \oplus \Gamma_2$ will next be selected along with Γ_3. If $|\textsc{prune}(\Gamma_1 \oplus \Gamma_2)|$ is larger than both Γ_3 and Γ_4, then the next cross-sum computed will be $\Gamma_3 \oplus \Gamma_4$.

4.4.2 IP Analysis Preliminaries

In order to simplify the analysis, we must make some unrealistic assumptions which will end up leading to very loose worst-case complexity results. We note that more complicated, but tighter upper bounds are possible by parameterizing the sizes of the cross-sum results. Let $|\Gamma_n^a| = Q$, $|\mathcal{Z}| = Z$ and assume $\forall z, |\Gamma_n^{a,z}| = |\Gamma_{n-1}| = M$. The latter assumption is somewhat reasonable and also conservative, since we do know that $\forall z, |\overline{\Gamma}_n^{a,z}| = |\Gamma_{n-1}|$ and $\forall z, |\Gamma_n^{a,z}| \leq |\Gamma_{n-1}|$. Note that under these assumptions we have $Q \geq M$.

As outlined in Section 4.4.1, the order which we select the sets does matter, however, for simplicity we will assume that the order is predetermined. The worst case for the IP algorithm is when the cross-sum sets get as large as possible as quickly as possible. Because the final set size is Q, none of the intermediate sets can be larger than this quantity, since $|\textsc{prune}(A \oplus B)| \geq \max(|A|, |B|)$. Thus, the worst case would be when the first cross-sum produces a set of size Q and we use this set in all the subsequent cross-sums. Note that this is very pessimistic, since it may be that $|A||B| < Q$, in which case it is impossible for their cross-sum to be as large as Q. This shows that using the relative sizes of M and Q could yield a tighter, though more complex upper bounds.

Under the assumptions that the first cross-sum yields the largest possible set and that all $\Gamma_n^{a,z}$ are the same size, always choosing the resulting cross-sum set and another one of the $\Gamma_n^{a,z}$ sets is equivalent to both the IP-LL and

IP-SL variations.

In particular, we can write the complexity as

$$\mathrm{IP}_*(Q, M, Z) = \mathrm{CS}_*(M, M, Q) + \sum_{i=3}^{Z} \mathrm{CS}_*(M, Q, Q) \ ,$$

where $\mathrm{CS}_*(M, Q, Q)$ is the complexity of a particular cross-sum operation such as $\mathrm{NCS}_*(\cdot,\cdot,\cdot)$ or $\mathrm{RR}_*(\cdot,\cdot,\cdot)$. This shows the first cross-sum of two sets of size M, followed by $Z-2$ cross-sums with one of the $\Gamma_n^{a,z}$ sets and the worst scenario of a size Q set. Using this, a conservative upper-bound can be derived, though the algebra gets cumbersome. To further simplify, at the expense of being even more conservative we assume that the first cross-sum is also done using M and Q sized sets and we get the simpler expression

$$\mathrm{IP}_*(Q, M, Z) = (Z-1)\mathrm{CS}_*(M, Q, Q) \ . \tag{4.3}$$

Total LPs

Using Equation 4.3 we can substitute in either $\mathrm{NCS}_{\mathrm{LP}}(\cdot,\cdot,\cdot)$ or $\mathrm{RR}_{\mathrm{LP}}(\cdot,\cdot,\cdot)$, but as discussed in Section 4.3.3 all cross-sum variations will require the same number of LPs (ignoring the issues to be discussed in Section 4.7.1). This is

$$\begin{aligned} \mathrm{IP}_{\mathrm{LP}}(Q, M, Z) &= (Z-1)\mathrm{GCS}_{\mathrm{LP}}(M, Q, Q) \\ &= (Z-1)MQ \ , \end{aligned} \tag{4.4}$$

which, using O-notation, is $O(ZMQ)$.

4.4.3 Normal Incremental Pruning

We begin our analysis with the simplest variation of the the IP algorithm for constructing Γ_n^a, which always chooses $\Delta = \widehat{D}$ for the cross-sum operations.

As mentioned, choosing this is equivalent to PRUNE$(A \oplus B)$. This is simply using the NCS cross-sums in the IP algorithm, so we define this variation of IP as *normal incremental pruning* (IP-NCS). We assume that we have already constructed all the $\Gamma_n^{a,z}$ sets and that they are parsimonious.

Total Constraints - Worst case

The regular IP variation uses the NCS algorithm, and we get

$$\begin{aligned} \text{IP-NCS}_{\text{worst}}(Q, M, Z) &= (Z-1)\text{NCS}_{\text{worst}}(M, Q, Q) \\ &= (Z-1)\left((Q+1)(MQ - \frac{1}{2}Q)\right) \\ &= (Z-1)\left(M - \frac{1}{2}\right)(Q+1)Q \ , \end{aligned} \tag{4.5}$$

which is $O(ZMQ^2)$.

Total Constraints - Best case

$$\begin{aligned} \text{IP-NCS}_{\text{best}}(Q, M, Z) &= (Z-1)\text{NCS}_{\text{best}}(M, Q, Q) \\ &= (Z-1)\left(2MQ + \frac{1}{2}Q(Q-3)\right) \\ &= (Z-1)\left(2M + \frac{1}{2}(Q-3)\right)Q \ , \end{aligned} \tag{4.6}$$

which is $O(ZQ^2)$, since $Q \geq M$.

4.4.4 Restricted Region Incremental Pruning

Recall that GIP is merely the IP algorithm where the generalized cross-sum algorithm is used. As previously discussed, analyzing the GCS algorithm directly is difficult since it depends upon keeping the history of the previous iterations to know which set would be chosen. However, we can bound the GCS total constraints by the minimum of the NCS and RR cross-sum variations. In this section we analyze a variation of the IP algorithm which always uses RR for the cross-sum operations and refer to this variation as the *restricted region* incremental pruning (IP-RR) algorithm. This will be used as the upper bound on the GIP algorithm.

Total Constraints - Worst case

In the worst case we have

$$\begin{aligned}\text{IP-RR}_{\text{worst}}(Q,M,Z) &= (Z-1)\text{RR}_{\text{worst}}(M,Q,Q)\\ &= (Z-1)\left(Q^2+M^2Q-\frac{Q}{2}\left[\frac{Q}{M}-1\right]\right)\\ &= (Z-1)\left(\left(1-\frac{1}{2M}\right)Q+M^2+\frac{1}{2}\right)Q\ , \qquad (4.7)\end{aligned}$$

which is $O(ZM^2Q+ZQ^2)$.

Total Constraints - Best case

The best case total constraints is

$$\begin{aligned}\text{IP-RR}_{\text{best}}(Q,M,Z) &= (Z-1)\text{RR}_{\text{best}}(M,Q,Q)\\ &= (Z-1)\left(M^2Q+MQ+\frac{Q}{2}\left(\frac{Q}{M}-3\right)\right)\\ &= (Z-1)\left(M^2+M+\frac{1}{2}\left(\frac{Q}{M}-3\right)\right)Q\ , \qquad (4.8)\end{aligned}$$

which asymptotically is $O(ZM^2Q+ZQ^2/M)$.

4.4.5 Generalized Incremental Pruning

Without being able to exactly characterize the generalization of the cross-sum operation (GCS), we cannot precisely analyze the fully general GIP algorithm. However, as previously discussed, the minimum of IP-NCS and IP-RR provides an upper bound on the GIP algorithm.

4.5 Witness

Recall from Section 3.2 and Table 3.7 that the witness algorithm is predominantly a loop over agenda items. For each iteration of the loop there are only two possible outcomes; either an item from the agenda is discarded, or we find a new vector to add to $\widehat{\Gamma}$, which is the set of vectors of Γ_n^a found thus far. As in the incremental pruning analysis we let $|\Gamma_n^a| = Q$, $|\mathcal{Z}| = Z$ and $|\Gamma_n^{a,z}| = |\Gamma_{n-1}| = M$ for all observations z. When we add a vector to Γ_n^a, we also add all its neighbors to the agenda, which amounts to adding $Z(M-1)$ items to the agenda. Thus the total number of agenda items that must be processed is $QZ(M-1)$.

Total LPs

Since each iteration of the witness algorithm's loop requires an LP, the total number of LPs required is the number of times the loop is executed. Since each iteration either removes an agenda item or adds a vector to $\widehat{\Gamma}$, the total number of LPs is the total number of agenda items plus one for each vector added to $\widehat{\Gamma}$. This is given by

$$\text{WITNESS}_{\text{LP}}(Q, M, Z) = QZ(M-1) + Q - 1 \quad , \tag{4.9}$$

where we only need $Q-1$ LPs for building up $\widehat{\Gamma}$, since $\widehat{\Gamma}$ is initialized with a vector prior to the execution of the loop. Asymptotically, the witness algorithm requires $O(ZMQ)$ LPs.

Total Constraints - Worst case

The worst case analysis for the total number of constraints for the witness algorithm requires an approach similar to the analysis for the PRUNE algorithm. There is a sequence of LPs to be solved and the sizes of these LPs are monotonically non-decreasing. The worst case is where the LPs get as large as possible as quickly as possible. For this to happen in the witness algorithm requires completely constructing Γ_n^a before removing any items from the agenda. Thus the first $Q-1$ LPs result in the addition of a vector to $\widehat{\Gamma}$ after which the agenda will contain all of the $QZ(M-1)$ possible agenda items.

Since $\widehat{\Gamma}$ is initialized with a vector, the first LP requires 2 constraints, comparing an agenda item against the initial vector added to $\widehat{\Gamma}$ and the simplex constraint. Since each of the first $Q-1$ LPs are adding to $\widehat{\Gamma}$, each LP will have one more constraint than the previous, up until $|\widehat{\Gamma}| = Q-1$. Note that the LP that finds the last useful vector will do so when $|\widehat{\Gamma}| = Q-1$, thus the first LP done when $|\widehat{\Gamma}| = Q$ is actually done for the first agenda item removed. Finally, to remove each item from the agenda requires an LP with $Q+1$ constraints and we have

$$\begin{aligned} \mathrm{WITNESS}_{\mathrm{worst}}(Q,M,Z) &= \sum_{i=1}^{Q-1}(i+1) + QZ(M-1)(Q+1) \\ &= \frac{1}{2}Q(Q-1) + Q - 1 + QZ(M-1)(Q+1) \\ &= (Q-1)\left(\frac{Q}{2}+1\right) + QZ(M-1)(Q+1) \ , \end{aligned} \tag{4.10}$$

which is $O(ZMQ^2)$. Note that similar to the PRUNE algorithm, the probability of this worst case result is relatively small.

Total Constraints - Best case

Inverting the worst case total constraint argument, the best case is where the sizes of the LPs stay as small as possible for as long as possible. For the witness algorithm, this is a little more complicated than the arguments used for the best case of the PRUNE algorithm. Recall from the PRUNE analysis on Page 116 that we assumed that all the useless vectors were removed before we found a second useful vector. A similar argument, applied to the witness algorithm, would say that we removed all the $QZ(M-1)$ agenda items while there was only a single vector in $\widehat{\Gamma}$. However, agenda items are only added when we find vectors to add to $\widehat{\Gamma}$, so unless the size of $\widehat{\Gamma}$ increases, we cannot process every item from the agenda.

To capture the interaction between the size of $\widehat{\Gamma}$ and the number of items on the agenda, for the best case we assume that we always find the next vector to add to $\widehat{\Gamma}$ with the very last item currently on the agenda. Thus the general progression is

1. add a vector to $\widehat{\Gamma}$,
2. add the $Z(M-1)$ neighbors of the vector to the agenda,
3. use an LP with $|\widehat{\Gamma}|+1$ constraints to remove each of $Z(M-1)-1$ items from the agenda,
4. use an LP with $|\widehat{\Gamma}|+1$ constraints to find a vector to add to $\widehat{\Gamma}$, and finally

5. repeat.

There are three things to slightly complicate this: the initial vector added to $\widehat{\Gamma}$, the final vector added to $\widehat{\Gamma}$ and, the most subtle complication, the agenda item used to find the next vector to add to $\widehat{\Gamma}$ is not immediately removed from the agenda, which then requires another LP to remove this item. Algebraically this progression of events and all of these complications combined give the expression

$$\begin{aligned}
\text{WITNESS}_{\text{best}}(Q,M,Z) &= 2Z(M-1) + \sum_{i=1}^{Q-1}(i+2)\left(Z(M-1)+1\right) \\
&= 2Z(M-1) + \left(Z(M-1)+1\right)\left(\sum_{i=1}^{Q-1} i + \sum_{i=1}^{Q-1} 2\right) \\
&= 2Z(M-1) + \left(Z(M-1)+1\right)\left(\frac{1}{2}Q(Q-1) + 2(Q-1)\right) \\
&= 2Z(M-1) + \left(Z(M-1)+1\right)(Q-1)\left(\frac{Q}{2}+2\right) \quad .
\end{aligned} \tag{4.11}$$

Asymptotical is $O(ZMQ^2)$, which is no different than its worst case.

4.6 Two-Pass

We analyze Sondik's two-pass algorithm because it has the potential to be nearly as effective as the IP and witness algorithms, based upon a best case analysis. Unfortunately, its worst case analysis is significantly worse, due to the problem of the imposter vectors possibly causing $|\widehat{\Gamma}| > |\Gamma_n^a|$ as discussed in Section 3.4.1.

In the worst case, when every vector in $\overline{\Gamma}_n^a$ that is not in Γ_n^a is an imposter, there would be $M^Z - Q$ imposter vectors. This would make the two-pass algorithm exponential in the number of observations, which is much worse that the IP and witness algorithm which are polynomial in $|\mathcal{S}|$, $|\mathcal{Z}|$, $|\Gamma_{n-1}|$ and $|\Gamma_n^a|$. However, if the value function has few or no imposter vectors, then the algorithm will not require an exponential amount of work and could prove to be competitive with the other algorithms. Since the two-pass algorithm has previously received little attention, here we provide some analysis to help characterized its performance and later provide some empirical results which show it to be a fairly effective solution procedure.

Total LPs - Best case

Assuming there are no imposter vectors, the number of LPs required by the two-pass algorithm is

$$\text{TWO-PASS}_{\text{LP}}(Q, M, Z) = QZ(M-1) \quad , \tag{4.12}$$

or $O(ZMQ)$. This results from having to check all the neighbors for each vector in Γ_n^a.

Total Constraints - Best case

Again, under the assumption that there are no imposter vectors, we get

$$\begin{aligned} \text{TWO-PASS}_{\text{best}}(Q, M, Z) &= QZ(M-1)\,(Z(M-1)+1) \\ &= QZ^2(M-1)^2 + QZ(M-1) \quad . \end{aligned} \tag{4.13}$$

This results from each LP having $Z(M-1)+1$ constraints and yields the asymptotic result $O(Z^2M^2Q)$.

4.7 Miscellaneous Issues

The previous analyses are based upon overly simplified version of the algorithms and there are many optimizations which can have a dramatic effect on their empirical performance. In this section, we briefly discuss some of these issues.

4.7.1 Saving LPs with Information States

All of the algorithms analyzed here have the general form of gradually building either a cross-sum result or the Γ_n^a sets. Some number of LPs are required to build this set up with some fraction of the LPs leading to a vector of interest. In many of these cases, the total number of LPs can be reduced by using a set of information states to initialize these sets. Recall from Section 3.1.2 that given a point, b, in information state space, generating $\gamma_n^a(b) \in \Gamma_n^a$ is trivial.

For specific examples:

- PRUNE(Γ) - given a set of information states, we could find the maximal vector from Γ, for each point, before resorting to LPs to check every vector. Here we save an LP for every unique vector we find from the set of points.

- NCS - this is just a PRUNE call, so the same savings can be used here.

- GCS - this is just a generalization of the PRUNE routine, so using a set of points to initialize the sets being constructed can be just as effective as in the PRUNE routine.

- IP/GIP - these are just the repeated application of the NCS or GIP algorithms and can utilize a set of points as described for these algorithms.
- **witness** - (also see Section 3.2.3) this algorithm uses an arbitrary information state to initialize the $\widehat{\Gamma}$ set. Similarly, any number of points could be used to initialize this set. Since each vector added to $\widehat{\Gamma}$ in the main witness loop requires one LP to discover, the more vectors initially put into $\widehat{\Gamma}$, the more LPs that are saved.

The issue of where to get such a suitable set of information state points is discussed below, with it breaking down into three basic approaches. The question of how this factors into the analysis of the various algorithms hinges upon what guarantees can be given concerning the possible number of maximal vectors that could be found using such a set. Using these guarantees, we can adjust the analysis of the algorithms to incorporate these savings. However, this mainly leads to more complex formulas and derivations without adding much to the general conclusions drawn from the analysis. Since extremely detailed analysis of these algorithms is not currently useful or enlightening, we have chosen to omit these more complex derivations.

Random Points

Generating information states at random is a particularly appealing method of generating a set of information points, since the number of points generated can be easily adjusted according to the amount of resources available. The drawback of this approach is that no theoretical guarantees can be made concerning the number of vectors that could be found using this approach. With some positive probability, the entire set of random information points could yield the same maximal vector. Note that properly generating a random information state, or more generally, a random probability distribution uniformly over the probability space is bit more complicated than the obvious approach of generating a random vector and normalizing. Appendix C discusses the proper method for generating random probability distributions.

Simplex Corner Points

The easiest set of information points to use are the simplex corners, which provides a few advantages over any other set of points that could be considered. First, when using one of these points to find a maximal vector in a set of vectors, we do not need to compute the full dot product $b \cdot \gamma$, since only one component of b is non-zero. Being able to reduce the computation to a simple component-wise comparison of the vectors makes this a particularly efficient set of points to use.

The other advantage is the guarantees that can be provided with the set of simplex corners. Suppose we have a non-parsimonious set $\overline{\Gamma}$ and we will apply the PRUNE to yield Γ using the simplex corners for the initialization. If all simplex corners are checked and the same vector in $\overline{\Gamma}$ is maximal for all of them, then we are guaranteed that $|\Gamma| = 1$ and we can skip the loop over the $|\overline{\Gamma}| - 1$ other vectors entirely. This follows from the convexity of the value function which Γ represents.

Further, if $|\Gamma| > 1$, then using the simplex corner checking is guaranteed to find at least 2 maximal vectors from $\overline{\Gamma}$. This gives a slightly better guarantee than is available for a random set of points.

Saving Points

The most intriguing and useful method for generating a good set of information points, as suggested by Littman [71], is to associate an information point with each vector maintained by the algorithms. We discuss how this is useful below, but note that this results in a slightly more complex implementation.

As an example of this approach, recall that the first step in all of the algorithms is to use Γ_{n-1} to construct the sets $\overline{\Gamma}_n^{a,z}$. One option, available to all the algorithms, is to prune this set to its parsimonious representation before applying any of the algorithms. The rationale for this approach is that each useless vector removed from $\overline{\Gamma}_n^{a,z}$ can result in a significant savings while using these sets in constructing Γ_n^a. In the analyses, the size of the $\overline{\Gamma}_n^{a,z}$ sets were assumed to all be M, which appears as a major contributor to the number and sizes of the LPs required by all the algorithms. Note that there is some amount of overhead required in pruning this set, and although the pruning will be relatively efficient in relation to the construction of the Γ_n^a set, it is still a non-trivial addition of LPs.

Suppose that we decide to always perform this initial pruning of the $\overline{\Gamma}_n^{a,z}$ sets. The pruning operation will process each vector and either find an empty region or return an information point lying in the vector's region. Normally, this point is ignored and discarded, but by saving this point we can use it in the individual operations of the algorithms to eliminate some of the required LPs.

For the incremental pruning algorithm we saw that for the individual cross-sum operations $|\text{PRUNE}(A \oplus B)| \geq \max(|A|, |B|)$. If all the parsimonious $\Gamma_n^{a,z}$ sets are used in the IP algorithm, then inductively we get the relationship

$$|\Gamma_n^a| \geq \max_z |\Gamma_n^{a,z}| \ , \tag{4.14}$$

which must hold regardless of the specific algorithm used.

We now focus on the problem of computing $A \oplus B$ where we have a saved a point for each of the $R(\alpha, A)$ and $R(\beta, B)$ regions, for all $\alpha \in A$ and $\beta \in B$. These points in the region are equivalent to the points we could have acquired in the pruning of the $\Gamma_n^{a,z}$ sets. Furthermore, assume that $|A| > |B|$. For the NCS algorithm, which is merely the PRUNE algorithm, using the set of points associated with A for the initialization of $\widehat{\Gamma}$ is guaranteed to find exactly $|A|$ maximal vectors; i.e., each point will yield a unique maximal vector from the full cross-sum. This saves $|A|$ LPs and using the points associated with B in addition to those points may yield even more savings, though we cannot guaranteed those additional points will have maximal vectors different from the set found with the points for A.

After initializing $\widehat{\Gamma}$ for this cross-sum pruning, the NCS (or GCS) routine will proceed to process the remaining vectors in the full cross-sum, uncovering those with non-empty regions to be added to $\widehat{\Gamma}$. Just as in the pruning of the $\Gamma_n^{a,z}$ sets, we do not want to throw away the information points uncovered for the useful vectors, but want to save them so that we can apply the same optimization in later cross-sum operations that might involve the results of $A \oplus B$.

The above description showed the basic idea behind using saved points to optimize the IP and GIP algorithms. However, this idea also applies to the initialization of the witness algorithm. The relationship of Equation 4.14 means that taking the points associated with the largest $\Gamma_n^{a,z}$ set will be guaranteed to produce $|\Gamma_n^{a,z}|$ distinct vectors, thus saving that many LPs. As with the cross-sum, using the points associated with the other $\Gamma_n^{a,z}$ may yield more vectors, but no guarantees can be made. Since the IP algorithm has guaranteed savings for each cross-sum, and the witness algorithm only has a guarantee on the final set size, it would seem that the saving of points would have a more dramatic effect on IP. We have done some preliminary experimental exploration into this issue, which demonstrated the effectiveness of saving points, but we have not yet completely characterized the savings or the differing effects on the various algorithms.

The analysis of the algorithms can be adjusted to incorporate this idea of saving points. Define PRUNE-SP(G, V, P) to represent the complexity of the PRUNE routine augmented with a set of points, such that for the given set of points it is guaranteed to find at least P distinct vectors. We then have

$$\begin{aligned}
\text{PRUNE-SP}_{\text{LP}}(G, V, P) &= G - P \\
\text{PRUNE-SP}_{\text{worst}}(G, V) &= \sum_{i=1}^{V-P} (P + i) + \sum_{i=1}^{G-V} (V + 1) \\
\text{PRUNE-SP}_{\text{best}}(G, V) &= \sum_{i=1}^{G-V} (P + 1) + \sum_{i=1}^{V-P} (P + i) \ .
\end{aligned}$$

Substituting these in for the complexity derivations for the NCS, GCS, IP and GIP analysis will yield the complexity for the variations of the algorithms which incorporate saving points.

The specific RR instance of GCS the equation to use for the number of LPs and total constraints is

$$\text{RR-SP}_*(M,N,C) = \sum_{i=1}^{M} \left[\text{PRUNE-SP}_*(N,C_i,1) + N(M-1)\right] \quad .$$

This case is a bit different from the others, since the region restriction means that we can only guarantee that we have one saved point lying in any region. In this case, there is some tension between the constraints saved by making $F \leq L$ and the number of LPs saved by making $F \geq L$.

As mentioned, this same idea applies to the witness algorithm, which under all the previous assumptions yields

$$\text{WITNESS-SP}_{\text{LP}}(Q,M,Z) = QZ(M-1) + Q - M$$
$$\text{WITNESS-SP}_{\text{worst}}(Q,M,Z) = \sum_{i=1}^{Q-M} (M+i) + QZ(M-1)(Q+1) \quad .$$

This assumes that the largest $\Gamma_n^{a,z}$ set is of size $|\Gamma_{n-1}| = M$.

Another possible variation of saving information points associates a point with each vector in Γ_{n-1}. At some point in all of the algorithms, to determine that a vector is a part of the true representation, an information point is produced. If we save these points, then on the next DP iteration, even if we decide not to do the extra work required to prune the $\overline{\Gamma}_{n-1}^{a,z}$ sets and generate points, we could use the set of points associated with the previous value function representation to initialize the various algorithms. This does not give any guarantees on the number of LPs that will be saved, but if the general structure of the PWLC function does not change too much from one iteration to the next, then the set of previous points might provide a reasonable amount of savings, with no extra computation, though a small amount of additional storage space.

4.7.2 Domination Checking

Section 3.1.1 discussed a simple, yet effective domination checking procedure to detect a useless vector without the need for an LP. Although it was presented in the context of optimizing the PRUNE routine, it can be incorporated into all of the exact algorithms. Any time an LP is set up to compare a vector against a set of vectors (e.g., the `findRegionPoint` routine), we can preface this with the simple domination check and forego the LP if the domination check shows the vector to be useless.

The problem with attempting to incorporate the domination checking into the analysis, is that there is no general way to know how many vectors may be removed from this simple check. One way to handle this in the analysis is to define some parameter representing the percentage of vectors which would be removed by the domination check.

For example, let $\Gamma = \text{PRUNE}(\overline{\Gamma})$, where $|\overline{\Gamma}| = G$ and $|\Gamma| = V$ and the pruning routine prefaces its main loop with a call to the domination checking routine. Let σG be the percentage of vectors remaining after the domination check. Then we have

$$\begin{aligned}\text{PRUNE-DOM}_{\text{LP}}(G, V) &= \sigma G \\ \text{PRUNE-DOM}_{\text{worst}}(G, V) &= \sum_{i=1}^{V} i + \sum_{i=1}^{\sigma G - V} (V + 1) \ .\end{aligned}$$

Doing this type of analysis for all the algorithms yields expressions which show the effects of the domination checks in terms of number and sizes of the LPs. This sort of detailed analysis is omitted here, but could be used to characterize the algorithms, and their variations, if future research can better characterize the σ parameter for problems, perhaps by analyzing the structure of specific POMDP instances. For varying values of σ, some algorithm variations may prove more effective than others.

We note that the domination checking does require a certain amount of overhead. If the check removes no vectors, then the effort is completely wasted, however the time required by the LPs dwarfs the time required by the domination checks.

4.8 Algorithm Comparisons

Given the closed form expressions derived in the previous section, we now have the groundwork for comparison of the various algorithms.

4.8.1 Cross-sum Comparisons

The most general cross-sum algorithm (GCS) has the flexibility to allow selection of the smallest comparison set, where the decision can be made as the algorithms progresses. This allows choices based upon the specific results of previous iterations of the main loop. However, this dependence on the previous iterations made the GCS algorithm difficult to analyze, since there is no a priori way to predict the results of these previous iterations.

The approach we used was to present the RR variation of the GCS algorithm, which always chooses the same set. Additionally, the NCS cross-sum algorithm is also a variation of the GCS algorithm except it uses a different predetermined set for the comparison (see Section 3.3.3). Because the

GCS algorithm has the freedom to choose and since we have analyzed two variations which do not choose, the complexity of the GCS algorithm for a given N, M and C is bounded above by the minimum of the NCS and RR algorithms.

We will compare the NCS and RR cross-sum algorithms, and see that for nearly all values of N, M and C, the RR algorithm has fewer total constraints in the worst case. This result will allow us to use the worst case total constraints for the IP-RR algorithm as an upper bound on the worst case complexity of the GIP algorithm instead of having to use the minimum of the IP and IP-RR algorithms, which is slightly more cumbersome.

We have seen that, without any of the possible optimizations, the total number of LPs required by both cross-sum variations is identical. Thus we restrict this comparison to the total number of constraints, which is equivalent to comparing the average sizes of the LPs.

Worst Case Comparisons

Comparing the worst cases of the two cross-sum algorithms is not completely conclusive, since we have no way of characterizing how likely these are to be achieved. However, since the GCS algorithm is bounded above by the minimum of the NCS and RR algorithms, the results of this analysis will allow us to almost always use the RR version as a bound on the worst case for the GCS algorithm.

Comparing the two cross-sum algorithms is equivalent to characterizing when the quantity

$$\mathrm{NCS}_{\mathrm{worst}}(N, M, C) - \mathrm{RR}_{\mathrm{worst}}(M, N, C)$$

or

$$NMC + NM - M^2N - NC - \frac{C^2(M-1)}{2M} - C$$

is greater than or less than zero. When this quantity is greater than zero, the RR cross-sum has fewer worst case constraints and is the preferred choice.

With the assumption that $N \geq M$, we have the constraints

$$M \leq N \leq C \leq MN \quad .$$

We can also restrict $M > 1$, since the result of a cross-sum of a single vector, α, and a parsimonious set, B is always $\{\alpha\} \oplus B$ with no pruning step required. Under these constraints, it can be shown that the NCS algorithm has a better worst case than the RR algorithm only in some very select cases. More specifically, these cases are:

- if $C = N = M$;
- if $C < 5$;
- if $C = 5$ and either
  - $N = 3$, $M = 2$ or
  - $N = 3$, $M = 3$ or
  - $N = 4$, $M = 4$;
- if $C = 6$ and $N = M = 5$.

Asymptotic Comparison

The better worst case bound for the RR algorithm for larger values of M, N and C can also be seen from an asymptotic analysis. We have the quantities

$$\begin{aligned} \mathrm{NCS}_{\mathrm{worst}}(N,M,C) &= NM(C+1) - \frac{1}{2}C(C+1) \\ \mathrm{RR}_{\mathrm{worst}}(M,N,C) &= M^2N + NC - \frac{C}{2}\left[\frac{C}{M} - 1\right] \quad , \end{aligned}$$

where the dominating quantity is $O(NMC - C^2/2)$ for NCS and $O(M^2N + C^2/2M)$ for RR. With the constraints on the sizes, and since these two quantities are not directly comparable, it helps to break these down into the two extreme cases $C = \Theta(N)$ and $C = \Theta(NM)$, this time using Θ-notation.

In the case $C = \Theta(N)$, NCS becomes $O(N^2M)$ and RR becomes $O(M^2N + N^2/2M)$. Since we have the constraint $N \geq M$, we see that RR has a better asymptotic complexity than NCS. The case where $N = M = C$ changes the analysis and results in NCS and RR both being $O(N^3)$, though NCS actually requires fewer constraints than RR. When $C = \Theta(NM)$ we get $O(N^2M^2)$ for NCS and $O(N^2M)$ for RR showing the clear preference for the RR algorithm when C is large.

4.8.2 IP vs. GIP vs. Witness

Total LPs Comparison

Asymptotically, the IP and witness algorithms are all $O(QMZ)$ for the total number of LPs required as seen from Equations 4.4 and 4.9. However, which algorithms requires fewer total LPs hinges upon the relationship between the size of Γ_{n-1} and the size of the observation set $\mathcal{Z}$, which are M and Z respectively. Essentially, if $M \geq Z$ the IP algorithms will do fewer LPs.

Total Constraint Comparison

Worst Case From Equations 4.5, 4.7 and 4.10 we get the following asymptotic results

$$\begin{aligned}\text{IP-NCS}_{\text{worst}}(Q,M,Z) &= O(ZMQ^2)\\ \text{IP-RR}_{\text{worst}}(Q,M,Z) &= O(ZQ^2 + ZM^2Q)\\ \text{WITNESS}_{\text{worst}}(Q,M,Z) &= O(ZMQ^2)\ ,\end{aligned}$$

Since we showed that RR was asymptotically better than NCS, it is not surprising that IP-RR is asymptotically better than IP-NCS. However, we see that the witness algorithm has the identical complexity to the IP-NCS algorithm, showing that the IP-RR algorithm is asymptotically the best exact algorithm for constructing Γ_n^a from Γ_{n-1} in terms of the total constraints required over all LPs.

Best Case To get some feel for the relative ranges on the total number of constraints we show the asymptotic behavior for the best cases of the algorithms. From Equations 4.6, 4.8 and 4.11 we get

$$\begin{aligned}\text{IP-NCS}_{\text{best}}(Q,M,Z) &= O(ZQ^2)\\ \text{IP-RR}_{\text{best}}(Q,M,Z) &= O(ZM^2Q + \frac{ZQ^2}{M})\\ \text{WITNESS}_{\text{best}}(Q,M,Z) &= O(ZMQ^2)\ ,\end{aligned}$$

We see that the best cases for the IP variants are slight improvements to the worst cases, while the witness algorithm's best case complexity is the same as its worst case complexity. This hints that empirically the IP variations could perform better than the worst case analysis would predict.

4.8.3 Two Pass

As discussed in Section 4.6, the worst case total constraints is exponential in Z for the two-pass algorithm due to the problem with imposter vectors. However, assuming that the imposter vectors are not a problem, the best case analysis makes the two-pass algorithm competitive with the other algorithms. In particular, the number of total LPs required is of the same asymptotic complexity as the other algorithms, namely $O(QMZ)$ (Equation 4.12). The complexity for the best case total number of constraints is $O(Z^2M^2Q)$ (Equation 4.13), which makes the two-pass algorithm polynomial in the parameters. However, this algorithm only has the potential to perform somewhere between the witness and IP algorithms.

4.9 Exact Empirical Results

Although the previous section's analysis helps to characterize the algorithmic variations and shows the asymptotic relationships between them, there is no better way to evaluate the the effectiveness of the algorithms than an actual comparison of the algorithms in execution. Many simplifying assumptions were made in the analysis section, which although always on the conservative side, may not have the same effects upon all the algorithms. Additionally, there are many factors that were ignored completely in the analysis; e.g. bookkeeping overhead, memory requirements, etc. Finally, there is a fair amount of overlap between the best and worst cases for the different algorithms. This section presents empirical results on a range of problems to serve as the evaluation of the previous analyses. We will use both randomly generated POMDPs and some small problems from the literature for which exact solutions are possible.

4.9.1 Random Problems

The complexity of solving POMDPs means that few researchers or commercial enterprises have embraced the model. As a result, POMDP *models* are a scarce commodity, despite the fact that POMDP *problems* are ubiquitous. Most of the existing POMDP models in the literature are fictitious domains which are either too small to glean useful conclusions when using them in comparisons, or too large to solve exactly. For instance, the majority of POMDP research papers in the operations research journals to date have examples with only 2 or 3 observations. An enumeration algorithm such as discussed in Section 3.3.1 can be quite effective on such problems, despite having best case complexity exponential in the number of observations.

In this section we explore randomly generated POMDP problems. These random problems allow us to tailor the experiments to a specific number of actions, states, observations or initial value function size. This will allow us to compare the exact algorithms by exploring the range for which they can practically solve the problems. For completeness, the next section will show empirical results on a few of the available POMDP models that are within the computational range of the exact algorithms.

Because our focus is to compare the individual algorithms for constructing Γ_n^a from Γ_{n-1}, it is not necessary to run value iteration for more than one step, and there is also little need to include the construction of Γ_n itself. Thus, we will predominantly be comparing the algorithms on the basis of constructing all the Γ_n^a sets from a given pseudo-random initial set of vectors representing Γ_{n-1}.

We first discuss our method for generating random problems and then present the empirical results using them.

Random POMDPs

Although one could imagine many definitions for random POMDPs we adopt one that is very simple.

Definition 4.9.1 *A* random POMDP *is a* POMDP *generated as follows:*

- *For each state-action pair, set the state transition function to be a random probability distribution chosen uniformly over all possible distributions.*

- *For each state-action pair, set the observation probabilities to be a random distribution chosen uniformly over all the possible distributions over the set of observations.*
- *For each state-action pair, set the immediate rewards to be a real number uniformly selected from a fixed interval. For all the problems presented, the interval is* [0 10].

Note that only the probabilities and rewards are random, and not the actual sizes of the sets, or ranges of reward values, that comprise the POMDP. We also do not generate the discount factor randomly, but assume that it is some fixed quantity. In fact, focusing on a single DP step will make the discount factor have negligible effect on the empirical result here and thus we used the discount factor $\rho = 1$.

At the heart of this definition is the generation of random probability distributions. Appendix C shows that the naive algorithm for doing this is not sufficient and presents the algorithm used in our empirical results for generating random distributions.

Random PWLC Functions

We have not devised any suitably satisfying definition for a random PWLC value function. However, for any definition, the inclusion of useless vectors should be avoided, since they do not contribute to the function is any meaningful way.

Thus we used a simple scheme to generate a fixed-sized parsimonious PWLC function by randomly generating vectors and throwing away useless or imposter vectors. This is repeated until a parsimonious set of the desired size is achieved. This suffers from many problems, the most prominent being that one relatively large vector, when added to a parsimonious set can yield a set much smaller than the original. This skews the PWLC function toward having large valued vectors. The practical concern of such a scheme is that much effort can be wasted generating useless vectors. Empirically, the computational problems only present themselves when trying to generate a large number of vectors for a problem with a small state space. For all the examples presented here, the size of the initial random representation for the input value function, Γ_{n-1}, was 10.

Defending Random Problems

There is often an expressed displeasure for random problems, since no one truly needs random problems solved. In many classes of problems, random problems tend to have nicer properties than those that really need to be solved and are a poor basis for empirical comparison of algorithms; in others random problems are harder than the "usual" problems. This objection tends to be less valid for the results presented here. By fixing the number of actions, states, observations and the size of the initial set of vectors, the only variable becomes the sizes of the Γ_n^a sets. However, given a specific input and output size, the running times of the algorithms are fairly predictable based on the number of additions and multiplications that must be performed. Thus, the actual values that are manipulated are of little consequence to the algorithms. Although this predictability would seem to negate the need for our empirical results, they are nonetheless important since the analysis ignored many of the implmentation and overhead requirements of the algorithms.

In fact, random problems tend to have very dense transition and observation matrices, where real problems usually exhibit some type of sparseness. If the sparseness is exploited, this would make random problems harder to solve than actual problems. Although not quantized, our experience has been that random problems of a given size are much harder to solve than an equivalent sized problem that is based upon more realistic system dynamics.

Having argued for random problems, we must also say that these algorithms are not completely immune to the concerns about random problems. All the algorithms have differing best and worst case complexity and it is unknown whether random problems are more likely to skew the algorithms in this range differently than realistic problems. For this reason, following the random problem results, we present some empirical results on problems that are loosely based on realistic domains.

Experimental Set-up

A problem instance is a particular size random POMDP and a particular initial value function. Although we generated many problem instances, every algorithm considered was run on every given problem instance.

Because even small random problems can require a significant amount of computation, we have had to impose an upper threshold on the running times. Any algorithm which takes more time than this threshold is terminated, and the threshold value is used as its running time. This is necessary, because imposing a threshold is an arbitrary scheme and once the threshold is met, there is no way to know whether it would have finished in the next millisecond, or the next millennium. By assuming it finished at the threshold value, we are adopting the most generous viewpoint. For our experiments, the particular threshold chosen was $1,800$ seconds.

As mentioned, we would run a single value iteration DP step and monitor the CPU execution time for the time spent in building the Γ_n^a sets. Because there is no dependence on the number of actions for these algorithms when comparing the construction of Γ_n^a, we have fixed the number of actions to be 4. Thus the execution time measures the time to construct four parsimonious sets from the initial set.

All of the algorithms are part of the same base of code and share a large percentage of the routines. This minimizes the amount of coding-dependent efficiency issues which might otherwise arise if completely separate implementations were compared. While none of the code is highly optimized, some care was given to ensuring that any slight coding optimization of one algorithm was equally considered for the other algorithms. There is still much savings that could be obtained from a detailed analysis of the routines for this code.

Because of machine precision issues and the way they interact with the various algorithms, it is possible for the algorithms to disagree about the size of the final set for a given problem instance. While this disagreement is rare, it is important to ensure it does not happen, since the complexity of the different algorithms depends upon the size of the resulting sets. We monitored the sizes of the answers for the various algorithms on each problems instance and for the approximately $2,000$ random problems generated all produced exactly the same sized solutions.

Because random problems of the same size can have varying complexity in terms of the sizes of the resulting Γ_n^a sets, we have averaged the execution times from a number of different instances of each initial problem size. All the experiments were run on the same architecture, Sun Ultra-Sparc 1, using the same operating system, Solaris, and the time is measured in CPU seconds using features of the operating system to track execution time of the individual process, rather than wall-clock time.

We have compared 4 algorithms: witness, IP-NCS, IP-RR and two-pass and have varied the number of states and observations in our comparisons. Some of the other optimizations were implemented, and are significant improvements to the basic algorithms, but no results for these are presented here. For all these algorithms, we used the initialization using the simplex corners discussed on Page 140.

Results

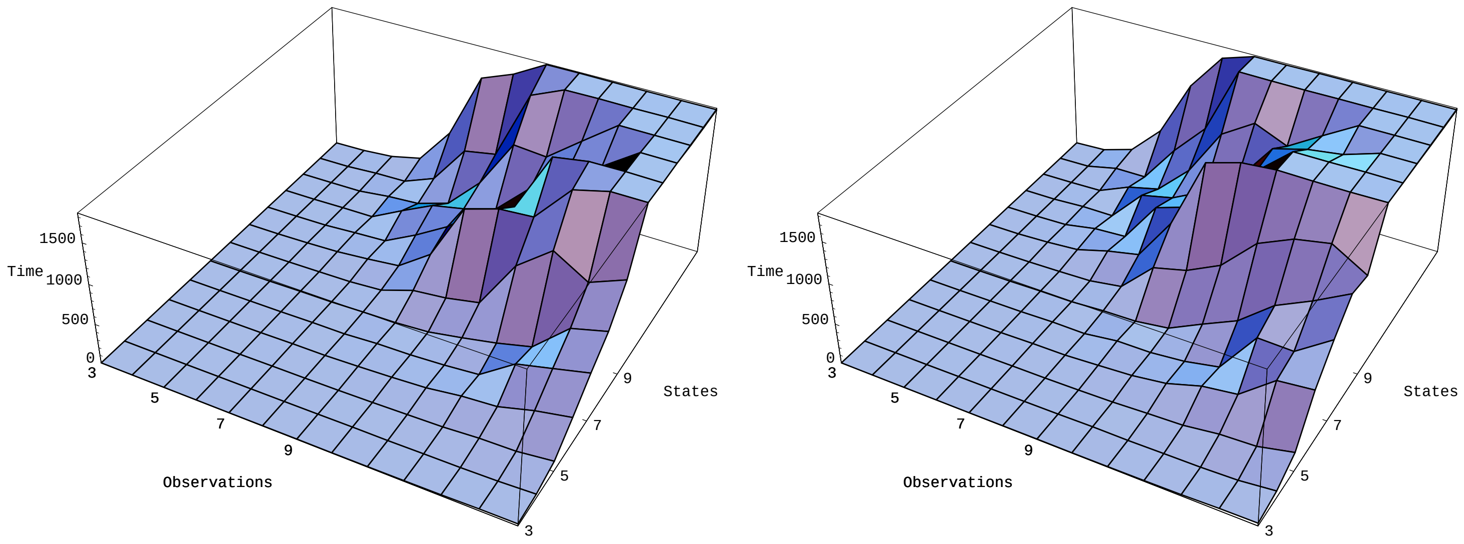
The three-dimensional plots in Figures 4.2 and 4.1 show the total execution times for constructing the four Γ_n^a sets as a surface over the axes representing the number of states and observations. In these figures the execution time for each instance of a problem size is an average of 5 different random problems. The differences here are not immediately noticeable, possibly with the exception of the witness algorithm, since these plots only tend to bring out the more dramatic differences. For this reason, we will look at some two-dimensional sections, which will also allow us to use a larger sample size. In the following graphs and tables, the results are the average over 25 different problem instances.

Figure 4.3 shows the amount of time required to construct the Γ_n^a sets when the state set size is fixed at $|\mathcal{S}| = 7$ and the number of observations varies from 3 to 15. We see what appear to be differences over the various algorithms, however, while these line graphs do give a nice characterization of performance, they mask the true relationship between the algorithms, since they hide the variances. Table 4.1 shows this same data in tabular form and shows the results of doing a simple two sample T-test. In this table, the best time is highlighted with a dark box around it, while the entries with lighter boxes indicate times which are not deemed significantly worse than the best time; i.e., the non-boxed entries are significantly worse. Since we have imposed a threshold on the execution time, there may be more significant differences than are shown in the tables.

We see that the two-pass algorithm is always the best, the witness algorithm is always significantly worse, and the two IP variants are sometimes competitive, especially for the smaller problems.

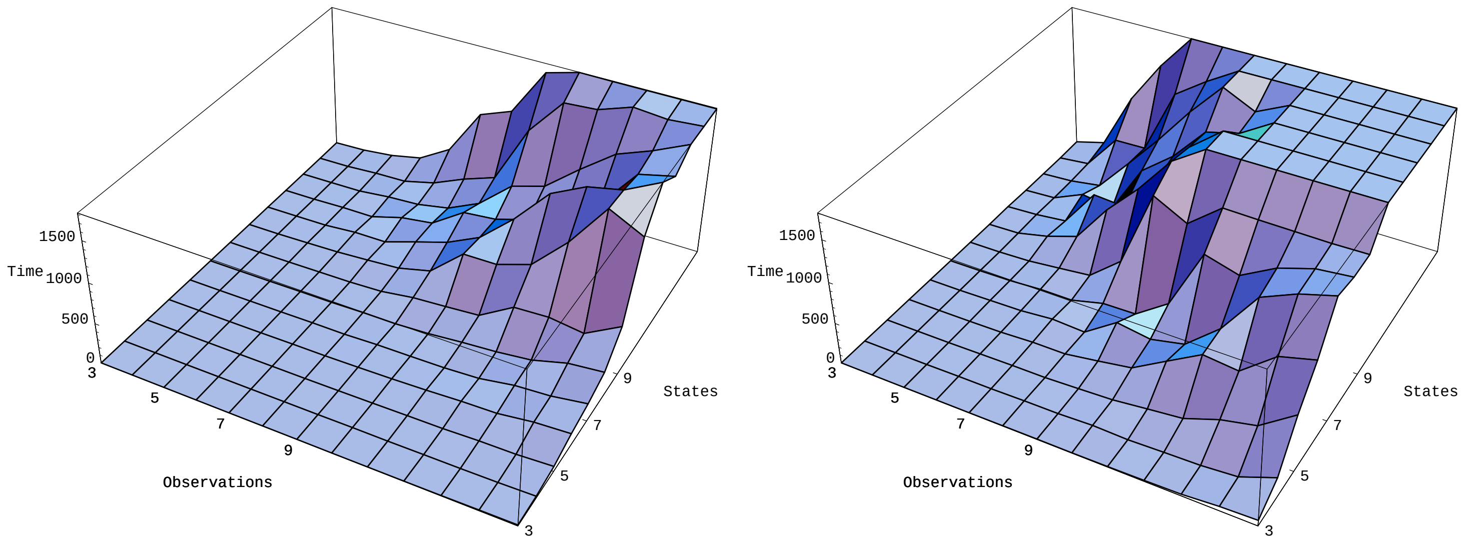
Figure 4.4 shows the results when the states vary from 3 to 15 and the observations are fixed at 7 with the corresponding Table 4.2 showing which differences are significant. We see that for varying numbers of observations, the two-pass and IP-RR algorithms start to dominate, with the two pass showing some significant differences.

Finally, Figure 4.5 and Table 4.3 shows the results when the the number of states and number of observations are varied simultaneously so that $|\mathcal{S}| = |\mathcal{Z}|$ at all times. Although visually, the differences between the algorithms is not as great, we see that the two-pass algorithm is the best, though this



(a) Restricted Region (b) Incremental Pruning

Figure 4.1:



(a) Two Pass

(b) Witness

Figure 4.2:

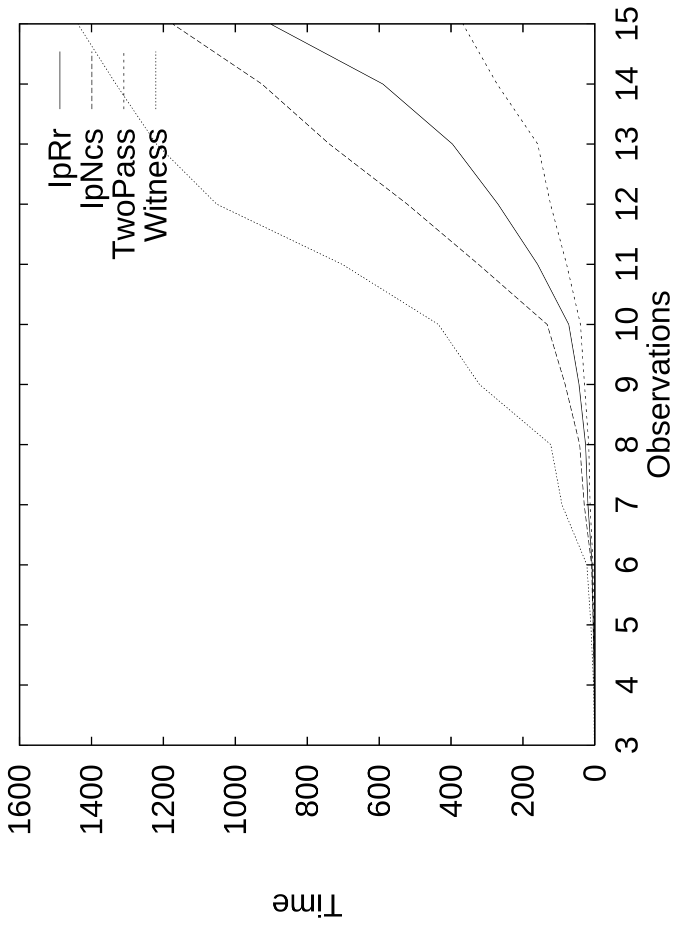


Figure 4.3: Total execution time for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{S}| = 7$.

Obs.	IpRr	IpNcs	TwoPass	Witness
3	$\boxed{0.729}$	$\boxed{0.591}$	$\boxed{0.589}$	1.315
4	$\boxed{1.784}$	$\boxed{1.681}$	$\boxed{1.292}$	3.534
5	$\boxed{4.391}$	4.880	$\boxed{3.052}$	11.069
6	7.667	9.187	$\boxed{5.360}$	22.253
7	19.278	29.527	$\boxed{13.361}$	91.578
8	26.003	42.387	$\boxed{16.958}$	122.502
9	$\boxed{43.928}$	82.499	$\boxed{28.896}$	320.850
10	72.610	132.690	$\boxed{39.928}$	435.048
11	$\boxed{159.196}$	323.672	$\boxed{78.634}$	702.685
12	269.402	521.814	$\boxed{123.418}$	1051.657
13	396.131	738.549	$\boxed{159.073}$	1219.628
14	589.646	925.986	$\boxed{271.939}$	1332.108
15	902.789	1174.707	$\boxed{367.210}$	1438.178

Table 4.1: Total execution time for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{S}| = 7$. T-test with $p = 0.95$.

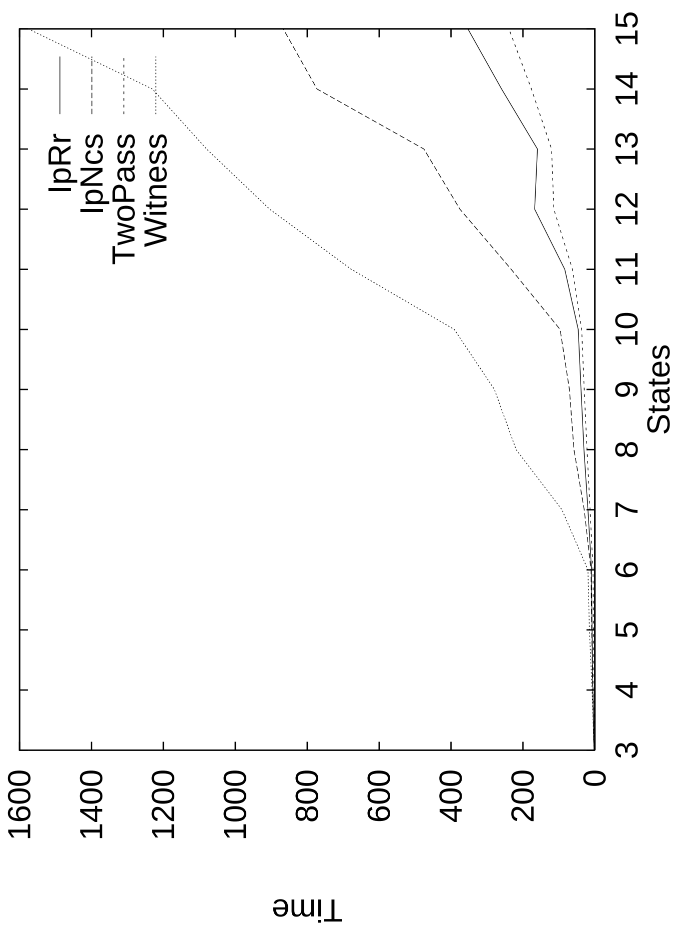


Figure 4.4: Total execution time for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{Z}| = 7$.

States.	IpRr	IpNcs	TwoPass	Witness
3	$\boxed{2.591}$	$\boxed{2.446}$	$\boxed{0.877}$	2.042
4	$\boxed{5.538}$	$\boxed{5.908}$	$\boxed{2.537}$	7.700
5	$\boxed{7.826}$	8.815	$\boxed{4.198}$	15.082
6	9.129	10.609	$\boxed{5.107}$	19.286
7	$\boxed{19.278}$	$\boxed{29.527}$	$\boxed{13.361}$	91.578
8	30.347	58.092	$\boxed{21.988}$	219.155
9	$\boxed{38.673}$	70.628	$\boxed{29.674}$	279.322
10	46.397	96.711	$\boxed{36.866}$	390.923
11	$\boxed{83.852}$	232.861	$\boxed{62.318}$	677.328
12	167.415	375.261	$\boxed{114.244}$	904.388
13	159.863	475.466	$\boxed{120.452}$	1079.734
14	259.331	772.531	$\boxed{176.225}$	1230.415
15	353.122	866.636	$\boxed{239.121}$	1576.024

Table 4.2: Total execution time for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{Z}| = 7$. T-test with $p = 0.95$.

States/Obs.	IpRr	IpNcs	TwoPass	Witness
3	0.328	0.268	$\boxed{0.181}$	0.343
4	1.014	0.901	$\boxed{0.523}$	1.163
5	2.870	2.875	$\boxed{1.675}$	4.545
6	5.737	6.563	$\boxed{3.586}$	12.036
7	19.278	29.527	$\boxed{13.361}$	91.578
8	62.840	126.947	$\boxed{40.211}$	513.418
9	239.923	606.104	$\boxed{136.525}$	1291.950
10	803.128	1194.624	$\boxed{384.270}$	1565.504
11	1484.407	1711.493	$\boxed{834.105}$	1798.577
12	$\boxed{1563.258}$	1652.665	$\boxed{1266.022}$	1689.086
13	1605.054	1797.613	$\boxed{1280.929}$	1803.345
14	1835.662	1821.193	$\boxed{1703.427}$	$\boxed{1803.323}$
15	$\boxed{1681.873}$	$\boxed{1671.918}$	$\boxed{1602.791}$	$\boxed{1670.595}$

Table 4.3: Total execution time for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{S}| = |\mathcal{Z}|$. T-test with $p = 0.95$.

is masked as the execution times of the algorithms reach the saturation point of $1,800$ seconds. Notice how quickly all the algorithms reach this saturation point when the problems sizes scale in both the observations and observations simultaneously.

Although the running time is what is ultimately of interest, our analysis focused on the numbers and sizes of the LPs. Figures 4.6 through 4.8 show the relationship between the algorithms in terms of the number of LPs solved for the three cases whose run times were shown above. Again, this is the number of LPs required just in building the Γ_n^a sets. As expected from the analysis, the witness algorithm requires more LPs that the IP variants. Although it isn't an asymptotic difference, the extra Q LPs (see Page 133)

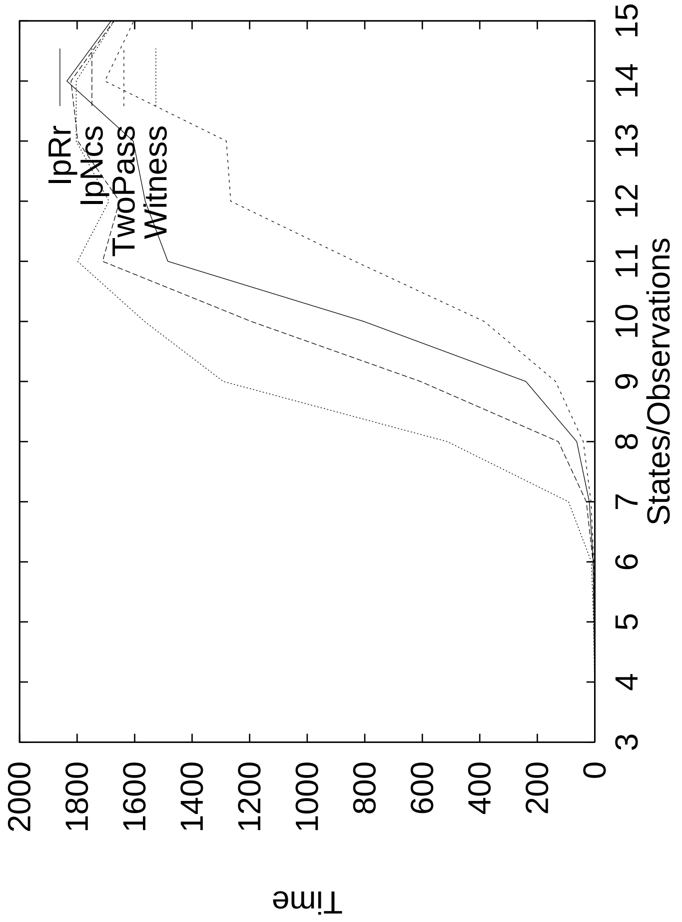


Figure 4.5: Total execution time for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{S}| = |\mathcal{Z}|$.

seems to be showing itself in these figures.

Perhaps most interesting is the large number of LPs required by the two-pass algorithm. We have already seen that the two-pass algorithm is among the fastest in these problems, yet seems to be doing far more LPs than the competitors. A partial explanation comes from looking at Figures 4.9 through 4.11 which show the total number of constraints required. Despite requiring many more LPs, the two pass algorithm is competitive in terms of total constraints, which means that the average LP size is much smaller than the other algorithm. It is unknown whether, in general, many small LPs are preferable to fewer large LPs and it may be highly dependent upon the actual constraint coefficients.

Although we have predominantly presented this empirical data pictorially, Appendix I.1 has the full numerical results with a T-test comparison and also includes data for the total running time, including the merging of the Γ^a sets.

Size Relationship between Γ_n^a and Γ_n

Recall from the theoretical analysis of Section 4.1.2 that the worst case complexity for constructing Γ_n from Γ_{n-1} is intractable. However, for the polynomial action-output bounded problems the witness and incremental pruning algorithms are tractable. Thus, there are problems where the Γ_n^a sets are exponentially larger than the Γ_n set. The interesting empirical question here is in exploring the relative sizes of the Γ_n^a and Γ_n sets. Using the experiments from the previous section and monitoring the size of the

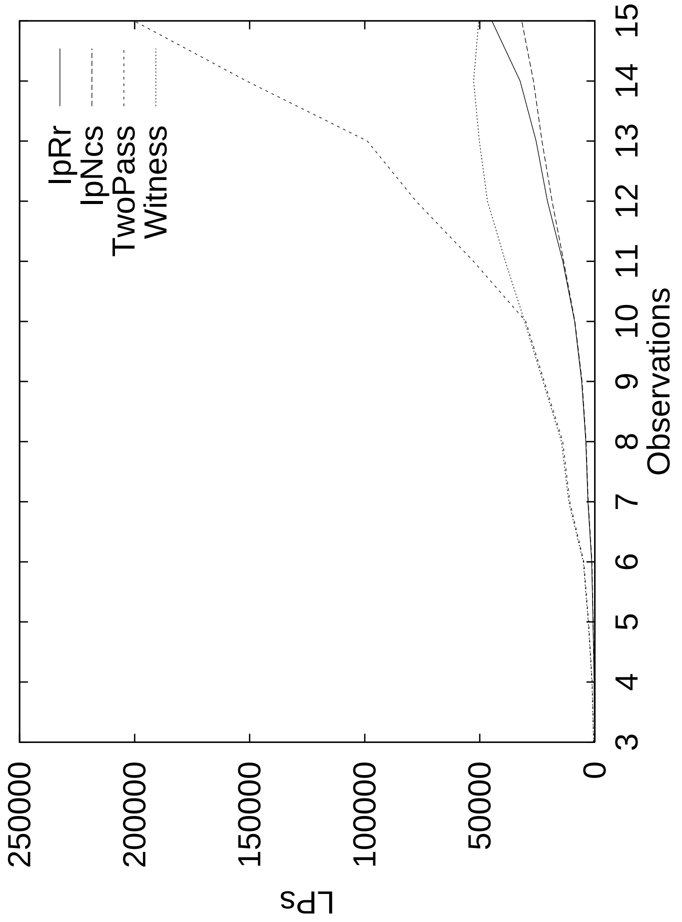


Figure 4.6: Total LPs for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{S}| = 7$.

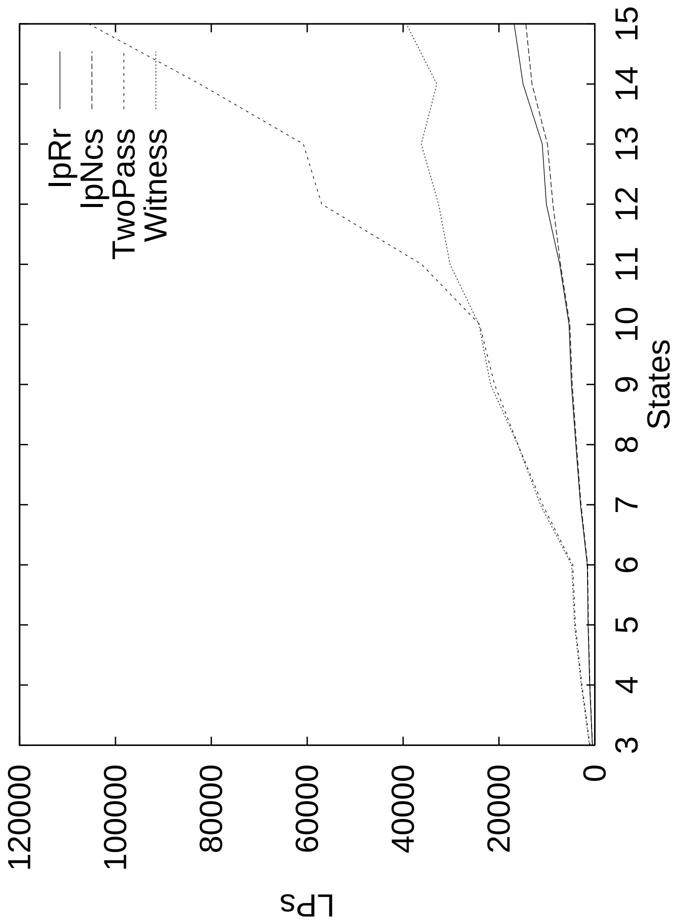


Figure 4.7: Total LPs for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{Z}| = 7$.

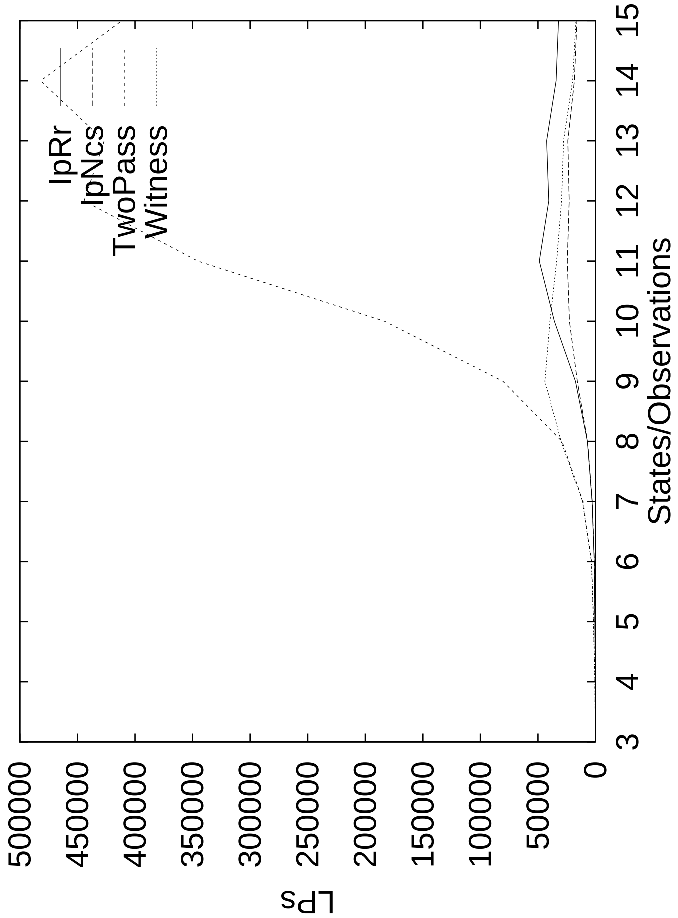


Figure 4.8: Total LPs for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{S}| = |\mathcal{Z}|$.

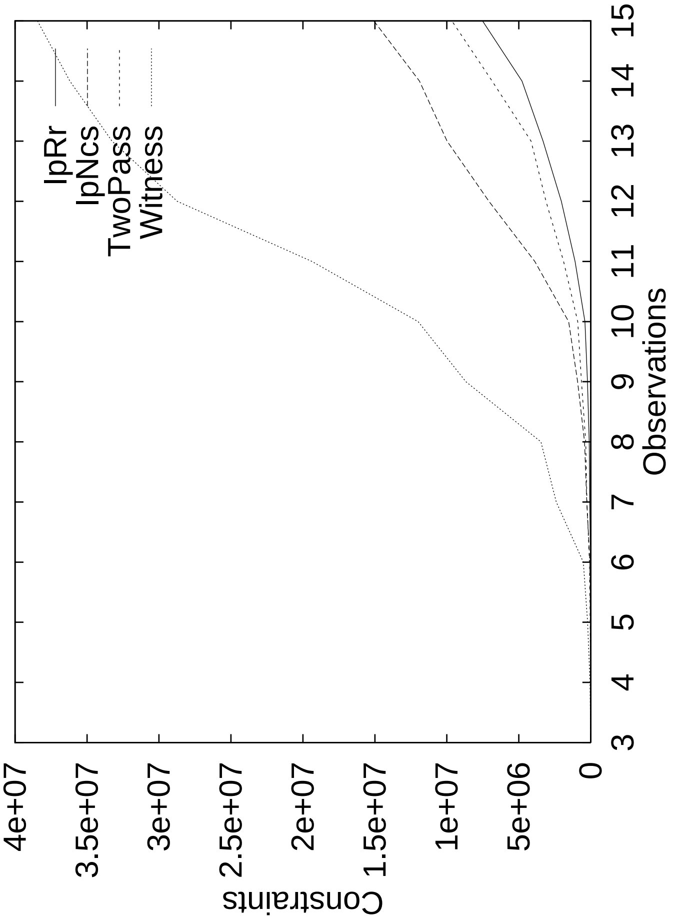


Figure 4.9: Total constraints for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{S}| = 7$.

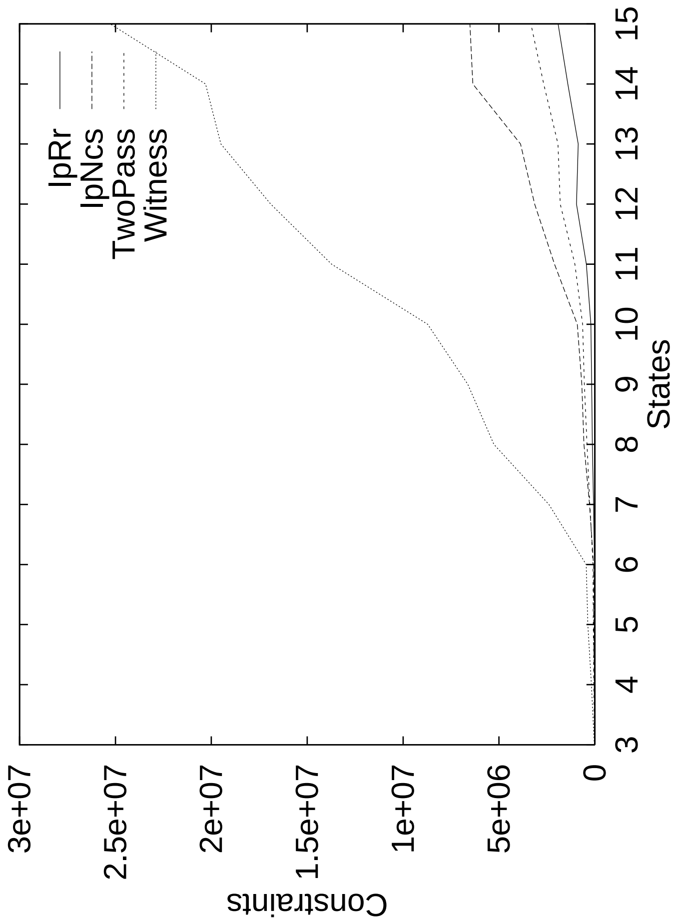


Figure 4.10: Total constraints for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{Z}| = 7$.

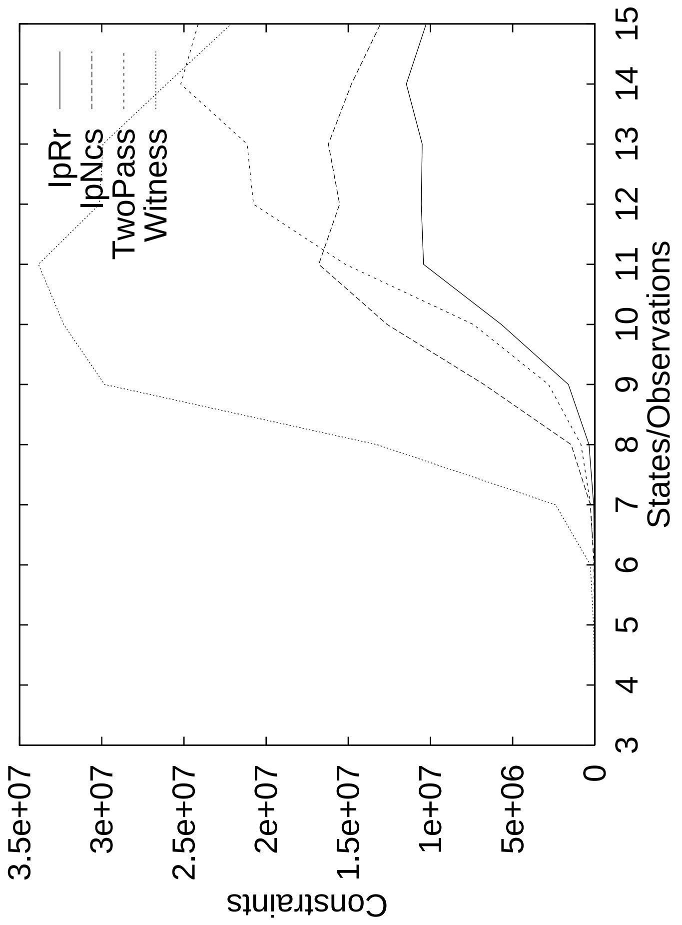


Figure 4.11: Total constraints for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{S}| = |\mathcal{Z}|$.

final set, Γ_n as well as the individual Γ_n^a sets, we computed the ratio

$$\frac{\sum_a |\Gamma_n^a|}{|\Gamma_n|}$$

For the experiments that were the basis of the three-dimensional plots in Figures 4.2 and 4.1, the average ratio was 2.559. Although there were a total of 845 individual POMDP problems[3], these statistics are based upon only 602 data points, since the runs which timed-out did not allow access to the final value function sizes. The variance for this case was 2.294 and the maximum ratio was 16.725.

We computed the same statistics for the empirical results shown in the line graphs and tables. In this case, based upon 687 data points, the mean was 2.438, the variance was 1.393 and the maximum ratio was 19.25. Thus, the worst case relationship between the sizes of Γ_n^a and Γ_n does not seem to occur on these random problems.

4.9.2 Small Problems

The purpose of presenting the algorithms on a set of small problems from the literature is two-fold: first, we want to alleviate any of the possible objections to the random problems of the previous section; second, we want to compare these algorithms in the more realistic setting of value iteration.

Experimental Set-up

For these algorithms we also imposed a threshold on the running time, but here used $3,600$ seconds. For a given problem instance, value iteration runs

[3] We ran 5 experiments for each problem size. The states and observations varied from 3 to 15 resulting in $13 \times 13 \times 5 = 845$ experiments.

all the algorithms with the execution time for each step of value iteration being monitored. If all algorithms completely solved the infinite horizon problem[4] then those results were reported. However, if all of the algorithms did not terminate, we used the number of value iteration steps of the algorithm that made the least progress for our point of comparison. Since we monitored the time for each iteration, we could report the time an algorithm took for that minimum number of iterations, even if it had progressed further.

We used 9 small problems which are listed in Table 4.4. This table shows the name of the domain, the sizes of the model for the domain, and the number of value iteration stages completed in the allocated time, the size of the final value function representation, Γ_f, and a reference to where this problem first appeared in the literature. Note that the SACI problem is a single aircraft identification problem similar to the IFF domain which is described in Appendix H.4.

We only executed each algorithm on each problem once. Since these are specific problem instances, all the algorithms operate deterministically. The only source of variance is in the functioning of the operating system. Because these algorithms are purely computational we expect, and our many experiences with these implementations show, that these variations are very minor. Therefore, for the tables presented in this section, the boxed entries in the tables serve only to highlight the best entry and are not indicative of statistical significance.

[4]We used a very conservative measure of the Bellman error between two successive iterations' value functions and stopped value iteration when it was below the threshold 10^{-9}.

Problem	States	Actions	Obs.	Stages	$\|V_f\|$	Reference
4x3	11	4	6	9	1375	[96]
4x4	16	4	2	374	20	[23]
CHEESE	11	4	7	373	14	[85]
PAINT	4	4	2	23	90	[62]
SHUTTLE	8	3	5	8	991	[28]
TIGER	2	3	2	19	61	[23]
NETWORK	7	4	2	18	578	[24]
NONLIN	7	3	6	404	5	[96]
SACI	12	6	5	4	258	[24]

Table 4.4: Small problem sizes, parameters and references.

Table 4.5 shows the result of the four principle algorithms on these problems. This table shows the total execution time in seconds for value iteration for the number of stages indicated in Table 4.4. However, our analysis focused upon the complexity of constructing the Γ_n^a sets and compared the algorithms from this perspective. In fact, the execution time spent building and pruning the $\Gamma_n^{a,z}$ sets and the time spent merging the Γ_n^a sets into Γ_n should be nearly identical[5] for all of the algorithms, modulo some small variance due to operating system unpredictabilities.

This common amount of execution time tends to mask the true savings available from the various algorithms. For this reason, we also kept track of the amount of execution time spent only in constructing the Γ_n^a sets from the $\Gamma_n^{a,z}$ sets and this is shown in Table 4.6.

The most obvious result is that for none of the problems is the witness algorithm the best, though there are domains where it is competitive with the IP variations. However, there are also domains where the witness algorithm

[5] The PRUNE routine's running time depends upon the order in which the vectors are processed. Since each algorithm produces the set differently, different orderings of vectors are given to the PRUNE routine.

	4x3	4x4	CHEESE	PAINT	SHUTTLE	TIGER	NETWORK	NONLIN	SACI
IP-RR	$\boxed{271.16}$	$\boxed{120.08}$	$\boxed{38.44}$	475.65	$\boxed{108.95}$	92.60	$\boxed{293.40}$	$\boxed{3.01}$	$\boxed{30.35}$
IP-NCS	685.35	200.45	40.63	378.23	158.81	74.37	494.89	$\boxed{2.91}$	42.60
TWO-PASS	487.25	284.81	55.51	$\boxed{329.70}$	277.56	$\boxed{54.10}$	679.83	10.57	161.49
WITNESS	2467.07	378.22	64.96	559.53	1567.13	68.50	2895.79	10.57	71.82

Table 4.5: Execution time in seconds for constructing Γ_n.

	4X3	4X4	CHEESE	PAINT	SHUTTLE	TIGER	NETWORK	NONLIN	SACI
IP-RR	$\boxed{33.00}$	$\boxed{0.26}$	$\boxed{13.16}$	453.01	$\boxed{21.02}$	84.51	$\boxed{115.31}$	$\boxed{0.12}$	$\boxed{19.92}$
IP-NCS	444.14	82.05	15.41	355.38	71.28	66.35	325.82	$\boxed{0.12}$	31.88
TWO-PASS	216.19	163.86	30.24	$\boxed{307.76}$	164.03	$\boxed{45.12}$	502.23	7.50	125.03
WITNESS	2221.37	259.02	39.96	538.17	1479.11	60.78	2726.67	7.68	61.43

Table 4.6: Execution time in seconds for constructing all the Γ_n^a sets.

is orders of magnitude slower.

Although the IP-RR algorithm gives the best times in 6 of the 9 algorithms, there are instances where it is not significantly better than IP-NCS and even instances where IP-NCS is just as good. Thus the more general GIP algorithm would be at least as good as the minimum of these two and perhaps even faster.

There are instances where the two-pass algorithm gives better times than both IP versions, though both of these domains happen to have only 2 observations. Recall from the analysis that the two-pass algorithm is the only one with a complexity that has a Z^2 term; i.e., quadratic in the size of the observation set. Overall the two-pass algorithm is reasonably competitive with the other algorithms, which means that the troublesome imposter vectors may only be a theoretical worry.

Our analyses earlier in the chapter focused upon the complexity of the algorithms in terms of numbers and sizes of the LPs that were solved, which we argued should be closely correlated with the execution time. Tables 4.7 and 4.8 show the number of LPs and total constraints used in constructing the Γ_n^a sets from the $\Gamma_n^{a,z}$ sets. Aside from the 4X4 problem, only in the 4X3 domain does the IP-RR algorithm do significantly fewer total LPs than IP-NCS, and in the CHEESE domain IP-RR requires nearly twice as many LPs as IP-NCS.

The most glaring numbers in these tables are the zeroes for the 4X4 and NONLIN domains for the IP variants. This comes from the initialization procedure that uses the simplex corners to initialize the Γ_n^a sets. Recall from Page 140 that when this initialization yields a single vector, the actual size of the set must be 1.

Although the IP-NCS and IP-RR variants do equivalent numbers of LPs on many of the problems, this does not translate into doing a similar number of constraints. As an example, the most interesting result is for the NETWORK problem where IP-NCS does 24,948 LPs and IP-RR does 24,017. Although roughly the same, the disparity in the number of constraint is large: 5,194,613 to 764,836. This translates into the disparity in the execution time where IP-NCS requires more than twice the time.

	4x3	4x4	CHEESE	PAINT	SHUTTLE	TIGER	NETWORK	NONLIN	SACI
IP-RR	$\boxed{4{,}094}$	$\boxed{0}$	4,424	$\boxed{78{,}079}$	$\boxed{2{,}796}$	$\boxed{19{,}802}$	$\boxed{24{,}017}$	$\boxed{0}$	$\boxed{2{,}661}$
IP-NCS	14,082	13,334	$\boxed{2{,}580}$	79,693	3,248	20,242	24,948	$\boxed{0}$	3,514
TWO-PASS	185,207	133,244	9,596	345,386	162,942	85,364	515,654	2,392	117,340
WITNESS	177,704	142,129	10,702	316,517	142,507	62,528	506,543	2,392	20,076

Table 4.7: Total LPs for constructing all the Γ_n^a sets.

	4x3	4x4	CHEESE	PAINT	SHUTTLE	TIGER	NETWORK	NONLIN	SACI
IP-RR	$\boxed{129,708}$	$\boxed{0}$	11,060	$\boxed{6,072,200}$	$\boxed{91,748}$	$\boxed{893,608}$	$\boxed{764,836}$	$\boxed{0}$	$\boxed{63,758}$
IP-NCS	3,813,050	99,950	$\boxed{10,320}$	10,094,731	765,084	1,085,400	5,194,613	$\boxed{0}$	201,229
TWO-PASS	9,923,058	1,331,584	28,040	52,372,798	10,873,016	7,432,708	35,890,936	7,166	5,570,868
WITNESS	84,483,806	1,400,181	47,958	37,368,848	78,422,690	3,643,184	147,942,760	9,558	1,918,561

Table 4.8: Total constraints for constructing all the Γ_n^a sets.

Note that the algorithms actually proceed much further than 19 stages on the TIGER problem. However, due to disparities in the manner which the machine precision interacts with the various algorithms, the sizes of the representations start to diverge beyond this point. Since the running times are sensitive to this quantity, execution time comparisons past the 19 stage are not meaningful.

4.9.3 Other Algorithms

We discussed the existing algorithms that predated the witness algorithm such as Cheng's linear support and Sondik/Monahan's enumeration scheme with the Lark/White pruning idea (see Sections 3.4.4 and 3.3.1). We showed that these are both, in the worst case, exponential in one of the relevant quantities. Specifically, the linear support algorithm is exponential in the size of the state space and the enumeration algorithm is exponential in the size of the observation set. In this section we briefly compare witness to these two algorithms and show that the empirical performance exactly matches this analysis. These empirical results first appeared in Littman *et al* [74].

Ideally, Sondik's one-pass algorithm should be included in this comparison, but the previous empirical and analytical results on the two-pass algorithm led us not to undertake the complications with this implementation. We attempted to get Sondik's original code, but no copies seem to exist [119]. Also, Cheng's linear support algorithm is our own implementation where every effort at efficiency was made. Here too, the ideal scenario would be to use the author's code directly, but this too seem to be no longer available [27].

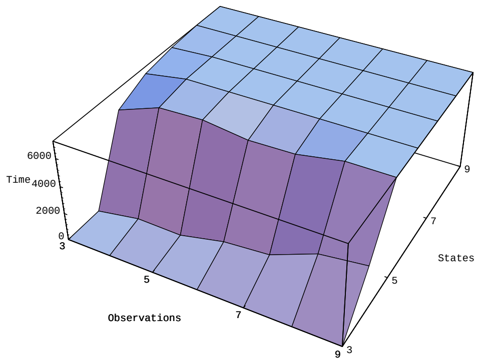
We present our results as three-dimensional plots in Figure 4.12, where the z-axis is the execution time in seconds and the x and y-axes show the effects of varying the numbers of states and observations.

As can be seen prominently in the figures, the linear support algorithm has an explosive increase in execution time as the number of states increases and the enumeration algorithm has a similar problem, except that it is sensitive to the number of observations.

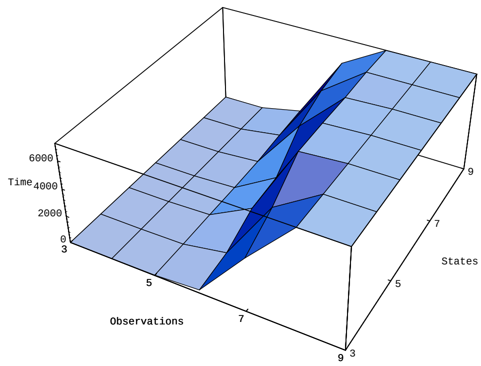
The plots of Figure 4.12 range over only 9 states and observations, since the other algorithms had already reached their saturation point. However the witness algorithm had not, so we ran some extra experiments out to 15 states and observations and combined them with the earlier results to yield a better picture of the witness algorithm's complexity, which we show in Figure 4.13.

The experimental set-up here is the same as for the random problems discussed on Page 156, except

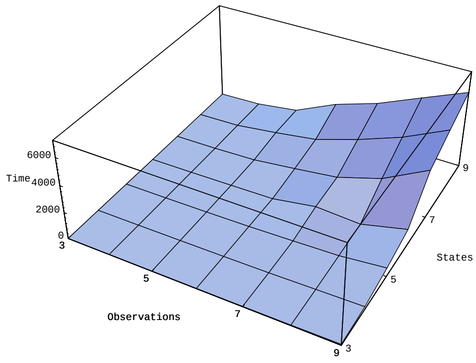
- the execution time is measuring time to construct Γ_n instead of Γ_n^a;
- the number of experiments averaged is 20 instead of 5; and
- the maximum execution allowed is $7,200$ instead of $1,800$.



(a) Linear Support



(b) Batch Enumeration



(c) Witness

Figure 4.12: Running times of the three algorithms over a range of POMDP random problem sizes.

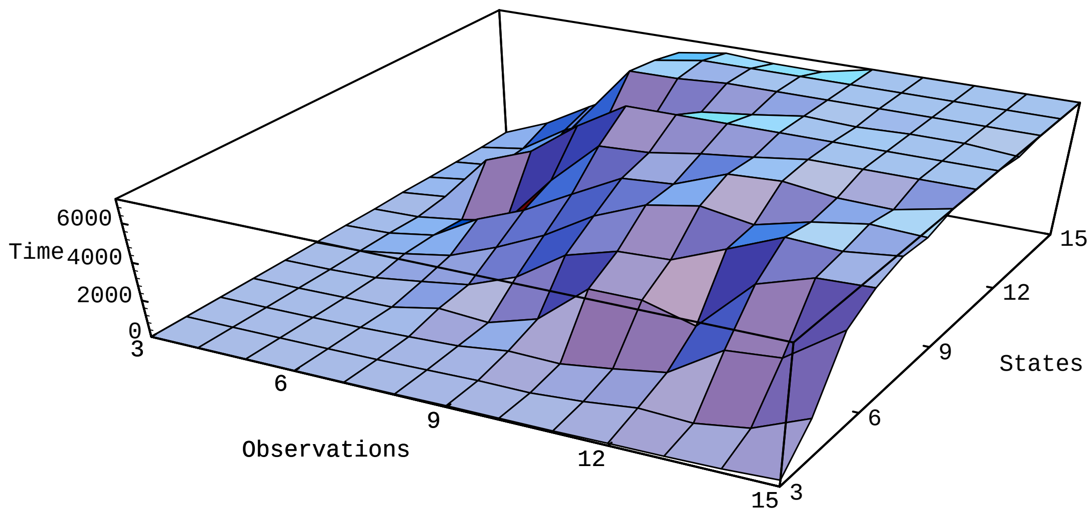


Figure 4.13: Running times of the witness algorithm over a larger range of POMDP sizes.

4.10 Conclusions

The orginal research in POMDPs and their algorithms was centered in the operations reseach community, where their focus was on developing the underlying mathematical theory. With the more recent interest in POMDPs from computer scientists, the issues of computational complexity and algorithmic development have come to focus. This perspective has resulted in better exact POMDP algorithms, but has also highlighted the limitations of such algorithms. Despite the theoretical limitations, there is still a place for effective exact algorithms, both from the algorithmic development viewpoint and as a basis for approximations. Additionally, empirical comparisons are crucial for algorithmic development, since worst-case complexity analysis does not always tell the whole story.

Prior to the mid-1980's, the empirical comparisons of POMDP algorithms were of very limited scope. The lack of realistic problems, combined with the restricted computing power of the day forced researchers to limit their experiments to problems with a handful of states and observations. Cheng [26] revisted the POMDP problem with some new algorithms and some of the most extensive empirical comparisons to date, but the problem sizes were still quite small and did not expose the real weakness with his schemes, which require enumerating all vertices of a convex polytope. He demonstrated that his linear support algorithm was more effective than Sondik's one-pass algorithm, but his conclusions only extend to the small state-set problems examined in his thesis. Although Sondik's one-pass algorithm has some theoretical complexity problems that need to be worked out, there is still an open question regarding the empirical effectiveness of this algorithm.

With the development of the witness algorithm, more extensive comparisons were undertaken [74] using a broader range of problem sizes. This has continued through the development and implementation of the incremental pruning and generalized incremental pruning algorithms [140, 24]. This chapter has presented both detailed analysis and empirical evaluations of the exact algorithms; a combination which has proven quite fruitful for the insight and development of this research, which has resulted in the current best exact POMDP algorithms, both in theory and in practice.

Chapter 5

Reinforcement Learning

In this chapter we will look at some applications of *reinforcement learning* (RL) algorithms [6, 53]. These techniques are a way to solve large, often continuous, state-space MDPs. Since a POMDP can be recast as a continuous state space MDP, these techniques provide a way to compute approximate solutions, although the algorithms that will ultimately result from this RL approach have no formal guarantees on the quality of their solutions. As discussed in Section 4.1.3 on the computational complexity of approximations, it is just as hard to find guaranteed approximate solutions for POMDPs as it is to find the optimal solution.

This chapter focuses on one particular approach, but there is much more work to be done in the area of applying reinforcement learning to partially observable domains. In the reinforcement learning literature these types of problems are referred to as ones with hidden state, incomplete perception, perceptual aliasing or non-Markovian environments. Also, in the RL parlance, our method would be referred to as a *direct* method, since we attampt to build the value function directly from experience. In an *indirect* method, the experience is used to construct a model, and the value function/policy is derived from the model.

Recently, Bertsekas and Tsitsiklis have used the name *neuro-dynamic programming* (NDP) in their attempt to connect the RL area to its mathematical basis in dynamic programming, function approximation and iterative stochastic approximations [11]. In this work refer to this class of techniques RL/NDP algorithms to reflect both their RL origins and the contributions of Bertsekas and Tsitsiklis.

RL/NDP is a framework for approximations in problems where solving directly would be intractable. For this thesis we will use it to derive approximate value functions (*approximations in value space*), though it is not infeasible to use these techniques to compute an approximate policies directly. There are two main ideas that distinguish the RL/NDP framework: first, to combat the *curse of dimensionality* [7], a function approximator is used; second, it uses simulated experiences to generate *trajectories* through the state-space, thereby avoiding explicit computations for all possible states and focusing the computational effort on the more likely parts of the state space.

For large problems the dimensionality of the state space is so large, or even continuous, that an explicit table look-up representation is not feasible. In the RL/NDP framework, a function approximator with a parameter space of lower dimensionality is used and is updated based upon simulated experiences. The updating of this function approximator typically uses some sort of *gradient decent* in parameter space, though more sophisticated optimization techniques are possible.

The RL/NDP framework is very general and allows any type of function approximator as well as many ways to generate and use the simulated experiences. This flexibility can be a drawback, since at this time there is not enough experience with these techniques to provide guidance for any particular problem or class of problems. On the other hand, since we are dealing with approximations, how well specific instances of an RL/NDP scheme work will depend on the specifics of the problems at hand. Different problems will require different choices, so the freedom in the RL/NDP framework is rich enough to allow a wide range of problems to be addressed.

The key to successfully applying RL/NDP techniques is to have some idea of the nature or structure of the problem being addressed. This will allow careful consideration of the options available and allow the RL/NDP instance to be tailored to best suit the application. However, this is often easier said than done and until there is more research into the the theory and applications of these techniques, this remains more of an art than a science.

The developed theory for RL/NDP concentrates predominantly on finite state spaces. Many of these ideas can be applied and extended to continuous state space problems, though the underlying theory is still in its early stages. For the problems we address, we have a continuous state space, so the theoretical guarantees we have are quite limited, thus we will rely heavily on empirical comparisons.

We begin by developing the basic RL/NDP framework and then show some specific instances that can be applied to yield approximate solutions to POMDP problems. We conclude this chapter with some empirical results applied to a suite of POMDP problems. We will be focusing on infinite horizon, discounted problems and because of our focus on approximations we will use $V^*(\cdot)$ for the optimal solution and $\widetilde{V}(\cdot)$ for an approximation of $V^*(\cdot)$. Since we will focus only on stationary policies, we will use $\pi(s)$ as the policy for state s rather than the decision rule $d(s)$.

5.1 RL/NDP Framework

At the one extreme we have the basic MDP dynamic programming based algorithms, such as value and policy iteration, which require an explicit model and an exact table look-up representation for the value function. The RL/NDP techniques are the other extreme of the DP spectrum using an approximate representation of the value function and using samples of the process instead of the full explicit model.

It is an involved process to systematically move from one extreme to the other and the book *Neuro-dynamic Programming* by Bertsekas and Tsitsiklis [11] shows this development in detail, as well and discussing many peripheral issues that arise along the way. In this section we follow Bertsekas and Tsitsiklis' exposition, but in much less detail and rigor, striving to give an overview with enough motivation for the techniques pursued for the POMDP problems considered here.

The development of the RL/NDP framework and the corresponding theory focuses on MDPs where the state space is finite. As the development progresses, the theoretical guarantees become scarcer and the final framework developed has no explicit restriction to finite state spaces. Although there are few theoretical guarantees for applying RL/NDP techniques in a continuous state space, we will see that some reasonable results are achievable.

Nonetheless, in our development of the RL/NDP framework we will make the finite state space assumption and all results referred to will implicitly be with respect to finite state spaces.

5.1.1 RL/NDP Outline

Before getting into the details of deriving the RL/NDP framework from its foundations in dynamic programming, we first provide a short outline which will serve as a road-map for the following subsections.

Asynchronous DP The basic value and policy iteration algorithms completely update the values or policy, iterating over the entire state space in a systematic order. The first step we take is to convert these algorithms to asynchronous versions where arbitrary ordering of the updates are allowed. This paves the way for simulation-based methods, where we do not enumerate the states, but simply sample from trajectories through the state space.

Function Approximation Most of the motivation for using simulations is that the state space is so large that we cannot or do not want to enumerate all states. When the state space is this large, we will not want to represent a value function over this entire space. Thus, the next step is to look at function approximation architectures for representing the value function.

These function approximators are parameterized with a parameter vector which is of lower dimensionality than the state space and are such that once the parameters are fixed, evaluating the value of a state is trivial. With the explicit value function, we can update the values directly, but with the function approximation, the problem becomes one of fitting the parameters of the approximator so that the values given by the approximator closely match those of the desired value. This becomes a non-linear unconstrained minimization problem, which is generally difficult to solve for a globally optimal solution. Thus a local iterative method is employed, though we must then sacrifice global optimality of the solution. These iterative methods are especially suited to simulation-based algorithms where sequences of samples are generated.

Stochastic Approximation Algorithms These local iterative methods are deterministic algorithms, assuming that a full data set of input-output pairs with the true values is available. Because we do not know the optimal value function to start and we cannot explicitly compute the full one-step value, we view the simulated experiences as samples from the full dynamic programming updates. We must then extend the iterative parameter adjustment algorithms to handle the case where there are stochastic samples which yields the general form for *stochastic iterative approximation algorithms.*

Simulation-based DP The stochastic iterative algorithms are exactly what is needed for the simulation-based methods with function approximators, but there are a few issues that arise when simulations are used to do dynamic programming even with an explicit table look-up value function representation. These issues are applicable to the function approximator case, but are best introduced without the added complications of function approximation.

Simulation-based DP with Function Approximation Having discussed the issues that arise with simulation-based DP with an exact value function representation, we are then ready to take the final step and replace the exact representation with a function approximator.

5.1.2 Asynchronous DP

The first step along the path to the full RL/NDP framework is to start with the basic DP-based value and policy iteration algorithms and adapt them so as to remove the specific dependence on a fixed ordering of states. We want to be able to select a state at random, do some value or policy update for this state and then return to select another state at random. We will ultimately be using simulations to select states, which will raise some concerns about the randomness of the states chosen, but we defer this issue until Section 5.1.5.

The value iteration algorithm from Section 2.2.3 has the following component update rule as its basis:

$$V(s) := \max_a \left[r(s,a) + \rho \sum_{s' \in \mathcal{S}} \tau(s,a,s') V(s') \right] \quad .$$

It will prove convenient here to use a more succinct operator notation so that we have $V(s) := (TV)(s)$ where the operator T transforms one value function into another such that component s of the transformed value function is

$$(TV)(s) = \max_a \left[r(s,a) + \rho \sum_{s'} \tau(s,a,s') V(s') \right] \quad .$$

With this operator notation, one step of the value iteration algorithm can be written compactly as $V := TV$.

If we simply embed the update $V(s) := (TV)(s)$ in a loop where s is selected randomly, we get the *asynchronous value iteration* algorithm. Although not necessarily an intuitive result, this algorithm will actually converge to the optimal value function, given that every state is selected infinitely often [8]. Note that this still requires a full model for the single-step value computation in the sum on s' and that we still need to explicitly store a value for each state. Although the "infinitely often" restriction may appear troublesome, even in practice, selecting each state often enough will yield a fairly good approximation to the optimal value function, though it is sensitive to the sampling process.

We can similarly adapt the policy iteration algorithm to an asynchronous version, but first we must alter the policy evaluation step presented in Section 2.2.4. As previously presented, the value of a given policy is computed by solving a system of equations. For large state spaces, such a procedure could be impractical, if not infeasible, and the common way large systems of equations are solved is through an iterative procedure.

The value iteration algorithm is an iterative procedure that can easily be adapted to solve the system of equations resulting from the policy value equations given by Equation 2.6. The only alteration of VI is the removal of the maximization over actions, so that the action chosen at each state update is simply the action specified by the current policy for that state. If we define the operator T_π such that component s is

$$(T_\pi V)(s) = r(s, \pi(s)) + \rho \sum_{s'} \tau(s, \pi(s), s')V(s') \ , \tag{5.1}$$

then the fixed-policy value iteration algorithm repeatedly calculates $V :=$

$T_\pi V$ and convergence to the value of policy π.

Since we can replace value iteration with its asynchronous variation, we get an asynchronous method for doing the policy evaluation step of PI. However, this in itself isn't enough for an asynchronous policy iteration algorithm, since we must also consider the policy improvement step. The basis of the policy improvement step is

$$\pi(s) := \operatorname*{argmax}_a \left[r(s,a) + \rho \sum_{s'} \tau(s,a,s') V(s') \right] \quad ,$$

where, in normal policy iteration, we iterate over all states s to compute the improved policy. For an asynchronous policy update we want to remove the restriction of having to iterate over all states.

The full *asynchronous policy iteration* algorithm is given in Table 5.1. In words, it says that we can arbitrarily select a state and then arbitrarily select whether to update its value, $V(s)$, or the policy for that state, $\pi(s)$. Using the full model for the one-step calculation and exactly representing the value function, this algorithm will converge to the optimal value function and policy, if each state has its value and policy action updated infinitely often *and* the initial value function and policy satisfy the condition $T_{\pi_0} V_0 \geq V_0$. Here V_0 is the initial value function, π_0 the initial policy and T_{π_0} is the one-step value iteration operator with fixed policy given by Equation 5.1

The most interesting aspect of the asynchronous policy iteration algorithm is that by selecting appropriate orderings of states and ordering of value/policy updates, many variations of algorithms can be constructed, including *Gauss-Seidel* value iteration, normal policy iteration, *modified policy iteration* [103] and others. For example, always following a policy

```
asychPolicyIteration(Ξ, ρ)
    π := any decision rule
    V := appropriate to satisfy T_π V ≥ V
    do
        s := Select state randomly
        if Update Value
            V(s) := (TV)(s)
        else
            π(s) := argmax_a [r(s, a) + ρ Σ_{s'} τ(s, a, s') V(s')]
    while Not Done
    return π
end asychPolicyIteration
```

Table 5.1: Code fragment for the asynchronous version of the policy iteration algorithm.

update with a value update for the same chosen state and selecting states at random, would yield the asynchronous value iteration algorithm.

These asynchronous algorithms will allow us to do DP with states generated through simulation where we update the value/policy as desired. We will later discuss the specific orderings and implications of interleaving the policy and value updates. However, note that while these asynchronous algorithms allow for arbitrarily selecting states, it is still required to have the full model and to enumerate and represent all possible next states.

When combined with function approximation, asycnronous value iteration takes the form of *actor-critic* schemes typically discussed in the reinforcement learning literature. The policy evaluation steps are viewed as a critic, judging the value of the policy that the actor is executing.

5.1.3 Function Approximation

The next step required to handle large state spaces is to avoid explicitly storing a value for each state. For this, a parameterized function approximator is used, where the dimensionality of the parameters is less than that of the state space. Additionally, it is assumed that once the parameters have been fixed, evaluating the value for a state is trivial.

Two of the major choices that must be made in developing an RL/NDP framework are determining the function approximator and an update rule for adjusting its parameters. These two choices are tightly coupled, though for the function approximators we consider here, the update rule takes on the same basic character of a steepest decent gradient method. This section discusses the update rule used in general and the later sections derive the specific update rules for the individual function approximators used for POMDPs.

The dynamic programming approach adjusts a state's value to make it closer to the optimal value, but with a function approximator we must adjust the parameters such that the the values given by the approximator are close to the desired values. Fitting the parameters to a specific function is an optimization problem; for the cases we consider it is a general non-linear least squares optimization problem.

Although we are ultimately interested in using simulation and immediate experiences to learn the optimal value function, we will start by describing the simpler problem of fitting an approximator to some known, fixed set of data, where each data point consists of an input (a state) and the desired output (the optimal value of the state). Although we will not know the optimal values for given inputs, it serves to illustrate the basic technique, and we will later show how this can be extended to be used in the RL/NDP framework.

Batch Gradient

In the more typical supervised learning applications in machine learning, a function is computed from a fixed set of training data consisting of a set of input-output pairs. If the set of input-output pairs are given by

$$\{(b_1, V^*(b_1)), (b_2, V^*(b_2)), \ldots, (b_M, V^*(b_M))\} \quad ,$$

and the parameters of the function approximator are given by a set Γ, we define $\widetilde{V}(\Gamma, b)$ as the value the approximator gives for input b and parameters Γ. The most often used optimization criteria for fitting an approximator to a function is to minimize the squared error,

$$\overline{e} = \frac{1}{2} \sum_{1 \leq m \leq M} e_m^2 \quad ,$$

where we define $e_m = V^*(b_m) - \widetilde{V}(\Gamma, b_m)$.

Since for general non-linear problems, performing this minimization is difficult, an iterative method is often applied. A common technique uses a form of *iterative gradient decent* where the parameter vector is continually updated according to

$$\Gamma_{n+1} := \Gamma_n + \eta_n d_n \quad ,$$

where η_n is a positive step-size and d_n is the *descent direction.* A common choice for d_n, and the only one we consider here, is the *steepest descent* where $d_n = -\nabla \overline{e}$ where the gradient is with respect to the parameters of the function approximator. This gives

$$\begin{aligned} d_n = -\nabla \overline{e} &= - \sum_{1 \leq m \leq M} \nabla e_m e_m \\ &= - \sum_{1 \leq m \leq M} \nabla \left(V^*(b_m) - \widetilde{V}(\Gamma_n, b_m) \right) e_m \\ &= \sum_{1 \leq m \leq M} \nabla \widetilde{V}(\Gamma_n, b_m) e_m \quad . \end{aligned}$$

This method is commonly referred to as the *batch* gradient method, since the decent direction is computed using the entire training set. This iterative procedure will convergence in parameter space, but the only guarantee about the point it converges to is that it is a stationary point of the error function; i.e., where the derivative is zero.

Incremental Gradient

Computing the descent direction over the entire data set is often undesirable, such as when the data set is large or, in the simulation context, since the simulated trajectories are not usually considered part of a fixed data set, but a steady flow of data. For both large data sets and simulation-based approaches, a commonly used technique, called *incremental gradient*, is to use only one item of the data set to compute the descent direction.

The pure incremental gradient method also assumes there is a fixed training set, but instead of using a sweep over the training pairs to generate a single descent direction, the parameters of the function approximator are adjusted after each input-output pair is processed during this sweep.

To show this more precisely, let the index n be the number of times we have iterated over the entire training set and m be the index of the particular training set pair being processed. Then the parameter update rule becomes

$$\begin{cases} \Gamma_{n,m+1} := \Gamma_{n,m} + \eta_n d_{n,m} & \text{if } m < M \\ \Gamma_{n+1,0} := \Gamma_{n,m} + \eta_n d_{n,m} & \text{if } m = M \end{cases} .$$

This variation always sweeps through the training set pairs in the same order, but another variation randomly orders the training set after each pass. For the incremental gradient method, the descent direction becomes

$$d_{n,m} = \nabla \widetilde{V}(\Gamma_{n,m}, b_m) e_m \quad .$$

Note that this technique can not properly be called a steepest descent method, since the gradient direction may actually be different from the true steepest descent direction.

This incremental variation of the gradient method can also be shown to converge by casting it as a regular gradient method with independent errors [11]. However, this incremental gradient method is still not flexible enough for the purposes of simulation-based dynamic programming for the following reasons:

- There is no properly viewed fixed data set.
- We do not have access to the true optimal values for a given b.
- The assumption of independence of the errors does not hold for the randomness generated by a Markov process.

The next section shows how the incremental gradient approach is a specific instance of a more general class of algorithms. Within this more general context, we will be able to rectify these problems with the incremental gradient method.

5.1.4 Stochastic Approximation Algorithms

The incremental gradient algorithm can be viewed a special case of a more general class of algorithms called *iterative stochastic approximation* algorithms. These algorithms form the basis of simulation-based dynamic programming (e.g., Q-learning [126], TD(λ) [121]) as well as for simulation-based gradient methods.

These iterative stochastic algorithms will allow us to remove the explicit summation over next states, by allowing us to take samples of the next states, which is exactly what will be required in the simulation context. We first present general deterministic iterative algorithms and see that value iteration is a specific instance of these. We then present stochastic versions of these iterative algorithms which serve as the basis for both simulation-based DP algorithms and incremental gradient methods. Finally, we show some specific instance of these for doing simulation-based DP where the method used to sample the process dictates the exact form of the iteration used.

Deterministic Iterative Algorithms

To derive the general form of an iterative stochastic approximation algorithm, we start by considering the simply case of trying to iteratively solve a system of equations

$$v = Hv \quad , \tag{5.2}$$

where the variables of interest are given by the vector v and H can be viewed as a coefficient matrix, or more generally, as an operator on the value vector.

For example, if we have the system of equations $Av = b$, then the operator H is such that $Hv = (A+I)v - b$ where I is the identity matrix. Alternatively, if we replace H with T_π, we get the system of equations for solving for the value of an MDP policy π.

Consider the simple deterministic iteration

$$v := Hv \quad .$$

In general, if the operator H has certain monotonicity or contraction properties, this iteration can be shown to converge to the solution of $v = Hv$. When we use the DP operator, $H = T_\pi$, we have $v := T_\pi v$ which is precisely the fixed-policy value iteration scheme for doing the policy evaluation step in PI. It can be shown that T_π and T both have the necessary contraction properties when the discount factor is $0 \leq \rho < 1$. We note that there are a class of MDPs where the transition probabilities impose a similar property on the DP operator even when $\rho = 1$. For simplicity, we restrict our attention to discounted problems.

If we have an iteration based upon individual component updating

$$v(s) := (Hv)(s) \quad ,$$

and $H = T$, then randomly sampling the states and using this iteration is equivalent to the fixed policy, asynchronous value iteration algorithm. Alternatively, this could be viewed as asynchronous policy iteration where there are no policy updates.

With a simple algebraic conversion, we can convert Equation 5.2 into an equivalent small step-size, η, version given by

$$v = (1 - \eta)v + \eta H v \ ,$$

and define an iteration based upon this as

$$v := (1 - \eta)v + \eta H v \tag{5.3}$$

with its single component version being

$$v(s) := (1 - \eta)v(s) + \eta(Hv)(s) \ .$$

This iteration is also valid and will converge when H has the necessary properties, though it is less useful than the normal iteration when the iteration involves a deterministic quantity. However, we will shortly be converting this to a stochastic version, where the small-step size will help reduce the algorithms sensitivity to the noise. Before introducing noise, we touch upon a few issues concerning the step-size.

Step-size Selection

Although not often used in practice, there are two assumptions regarding the step-size that are required for any theoretical convergence guarantees. Let n be the n^{th} iteration of the algorithm and η_n be the step-size used on that iteration, then the assumptions are

$$\sum_{n=0}^{\infty} \eta_n = \infty \tag{5.4}$$

and

$$\sum_{n=0}^{\infty} \eta_n^2 < \infty \ . \tag{5.5}$$

The first assumption is needed to ensure that regardless of the initial value, all possible values are reachable. If this assumption is violated, then the iteration can only move a fixed distance from the starting values. If the actual answer is further away from the initial value function than this distance, there is no way the algorithm could converge to it. The second assumption is required to ensure that the step-size goes to zero, which is needed if convergence to anything can be expected.

These step-size reductions are usually not adhered to in practice. The step-size has an important effect in the empirical convergence rate of many of the algorithms, so step-size adjustment schedules which give good empirical results are preferred to the more theoretically motivated restrictions. In addition, there is a reluctance of reducing the step-size to zero, since many RL/NDP algorithms want to allow adaptation to changing environments. We note that RL/NDP techniques only have convergence guarantees on stationary environments, but are nevertheless applied in dynamic environments with reasonable results.

Adding Noise

The step size variation is introduced because it becomes a more desirable iteration when we do not know H precisely or Hv is difficult to compute. For these cases, we prefer to sample Hv, as will be the case when we simulate trajectories through the state space.

Suppose we introduce a noise random variable w with zero mean. The the iteration above becomes

$$v := (1 - \eta)v + \eta\,(Hv + w) \quad , \tag{5.6}$$

where individual $Hv + w$ are noisy samples of Hv. This is the general form of a stochastic approximation algorithm.

The specific instances we will be concerned with here assume that we have a random variable s and a function $f(v, s)$ such that $E[f(v, s)] = Hv$. We view $f(v, s)$ as a sample of Hv which will depend upon s, which is a random quantity which will be driven by the simulation of trajectories through the state space.

We can rewrite Equation 5.3 as

$$v := (1 - \eta)v + \eta E\left[f(v, s)\right] \quad , \tag{5.7}$$

where here $E[f(v, s)] = Tv$. This is still a deterministic iteration and assumes we can compute this expectation exactly, which requires fully computing the one-step DP operator as in VI.

If we cannot compute the expectation, we could take a set of samples of Tv, compute the sample mean and use this in the iteration. Because there will be some sampling error, or noise in the computation of the expectation, this would become a stochastic approximation algorithm. However, as the sample size increases, the noise diminishes, the sample mean approaches the true mean and we progress closer and closer to the deterministic iteration. If we let the sample size be 1 we get

$$v := (1 - \eta)v + \eta f(v, s) \quad , \tag{5.8}$$

which is an iteration based upon a single sample, which is more generally known as a *Robbins-Monro* stochastic approximation algorithm and is a specific instance of the more general stochastic approximation algorithms.

These single sample stochastic approximation schemes will be the central approximation algorithm used for the RL/NDP techniques.

Gradient Descent as an Iterative Approximation

Although we focused our discussion of the iterative algorithms for the case when $H = T$ or $H = T_\pi$, if we let $Hv = v - \nabla f(v)$, then Equation 5.3 becomes the batch gradient algorithm by letting v be the parameter vector Γ and $f(\cdot)$ being the error function, $\overline{e}$. Thus, the batch gradient descent algorithm is a specific instance of a deterministic iterative algorithm with small step-size.

Consider the incremental gradient descent algorithm where we do not compute the full gradient $\nabla\overline{e}$. If we view the incremental updates as simply noise corrupted samples of $\nabla\overline{e}$, then the incremental gradient algorithm is nothing more than an instance of an iterative stochastic approximation algorithm based upon the equation $v = v - \nabla\overline{e}$.

Convergence

The convergence proofs for the stochastic approximation algorithms are quite complicated and we refer the reader to Bertsekas and Tsitsiklis [11] for a comprehensive treatment. However, in this work they show convergence for the cases of interest to the techniques described here; iterations based upon the dynamic programming operators and based upon gradient descent directions. However, there are many technical conditions which must hold for these to be valid. Under the right conditions, all of the following have been shown:

1. when H is a a contraction mapping as is the case for the DP operator;

2. when $Hv = v - \nabla f(v)$ which is when the updates are based upon descent directions as in the gradient descent algorithms; and

3. when the noise w is not independent from one iteration to the next, which occurs when samples are generated from a Markov process.

The first convergence result is only useful for simulation-based dynamic programming techniques where a full explicit value function over all states can be maintained. The second and third convergence results are of the one of most interest here, since we will ultimately be concerned with function approximators and sampling from a Markov process. The convergence results for the dependent noise case is specific to Markov noise and requires the most assumptions for the convergence properties to hold.

5.1.5 Simulation-based DP

The asynchronous DP methods require the sampling of the states to have certain properties, such as sampling each state infinitely often. In practice, this is not achievable and there is a need for a good sampling technique. Here we will use simulated trajectories through the state space as the sampling mechanism for the asynchronous DP methods. In this section we assume that we can explicitly represent the entire value function, since this will be a simpler context to discuss the issues that arise when using simulations to generate the states. This will leave us one step away from the full RL/NDP framework where the final step will be to add function approximation.

Sampling the Process

Let us first visit the case where we have a fixed policy π and are simply interested in using simulation-based, fixed policy, asynchronous value iteration. The iterative stochastic algorithm of Equation 5.6 where $H = T_\pi$ provides us with the algorithm

$$v(s) := (1 - \eta)v(s) + \eta \left(\left(T_\pi v \right)(s) + w \right) \quad .$$

The only unspecified item is what to use for the individual samples $(T_\pi v)(s) + w$. In this case, since $(T_\pi v)(s)$ is the true infinite horizon value of state s, one candidate sampling method would be to repeatedly start at state s, simulate the process, compute the discounted sum of rewards during this simulation and use all these samples to compute a sample mean. This would be equivalent to the noise-corrupted version of Equation 5.7 where the sample mean replaces the expected value. However, as mentioned, we will only concern ourselves with the single sample, Robbins-Monro variation shown in Equation 5.8, thus a sample would be a single trajectory starting from state s.

The first problem is that for all practical purposes will not be able to simulate an infinite trajectory. However, if the MDP has the property that it is always guaranteed to enter a state where no more costs will be accumulated, then upon entering this state (i.e., zero-cost absorbing state), the simulation can be stopped and that sample used. Problems with this type of structure are often referred to as *stochastic shortest path* problems. For general infinite horizon MDPs, we may never be able to reach a zero-cost absorbing state. Although any discounted infinite horizon MDP can be converted into a stochastic shortest path version [11], this is not usually desirable.

When zero-cost absorbing states are not available, some form of truncation of the trajectory is required. For any given discount factor, we can make the error due to truncating the trajectory as small as desired. By making the trajectory appropriately long, the error is negligible and we can assume we are using the entire trajectory. Unfortunately, for discount factors close to 1, this could be an extremely long trajectory.

Often the trajectory is truncated to a point that is computationally convenient rather than as a result of trying to get the error below some threshold. The parameter here is the length of the trajectory to use, and we note that even when we have an MDP with zero-cost absorbing states, we may want to apply this truncating technique. In the extreme case, we can consider truncating the trajectory to length 1, which is actually the case we consider in the specific POMDP algorithms presented later.

Another technique is to "discount" the effects of the rewards along the trajectory as the distance from the beginning of the trajectory increases. Although this seems similar to the discount factor we already have for the MDP, it is something entirely different. This parameter is serving to adjust the sample and not the optimal value function. We do not use such a technique in this work, but mention that this sample discounting parameter, λ, is basis of the TD(λ) approach [121]. Using the entire trajectory without any sample discounting is equivalent to TD(1).

Simulating a long trajectory starting from a single state and getting a single sample turns out to be very data inefficient. The trajectory visits many states along the path and we can view the sub-trajectory that starts from any of theses states as a sample trajectory from those intermediate states. A more data efficient technique would be to update the values of each state visited by the trajectory. The full TD(λ) approach combines updating every state along the trajectory with the discounting of the sample. There are some technical details concerning this combination of updating and sampling which needs to be considered, but which we ignore here.

The approach we focus on here is to use a series of simulated trajectories, update all states along the trajectory, but only use a single transition as the basis for our sample. This can also be viewed as TD(λ) with $\lambda = 0$.

Despite the myriad of options available for sampling the process, any choice of these used in the stochastic approximation algorithm will converge to the proper value function for that policy, under the right set of technical conditions.

Simulation-based Policy Iteration

The previous section shows that we can use simulation in a fixed-policy value iteration scheme and converge to the proper value function. This can form the basis of a simulation-based policy iteration algorithm, where the policy is evaluated using the simulation-based techniques of the previous section, then the policy evaluation phase is followed by a policy improvement step. Unfortunately, the policy improvement step seems to involve an explicit enumeration over all of the next states, which is contrary to the arguments for using simulation-based techniques.

One approach is to, again, use samples of the next states to compute the one-step improved policy. To do this we introduce the Q-functions for a policy, which represent the value of taking the immediate action a in a state s and following policy, π, thereafter. They are given by

$$V_\pi^a(s) = r(s,a) + \rho \sum_{s'} \tau(s,a,s') V_\pi(s') \ \ ,$$

and are precisely the quantity we need to compute in a policy improvement step. With the Q-functions we can then use simulation steps to compute these Q-functions and then do the policy improvement with

$$\pi(s) := \operatorname*{argmax}_a V_\pi^a(s) \ \ .$$

The problem with this approach is that in order for the new policy to actually be an improved policy, we must have the true Q-function values for each state-action pair. To do this would require evaluating the Q-functions for each state-action pair, infinitely often. Since this is practically unachievable, there is no way to guarantee that the new policy will be an improvement. Thus, a simulation-based policy iteration scheme of this sort is sensitive to the initial states chosen, since the initial states dictate the paths the trajectories take.

Simulation-based Value Iteration

The previous section outlined a policy iteration scheme which used a full simulation-based evaluation to find the value of a policy and then a full simulation-based scheme to update the policy via the Q-functions. Similar to asynchronous value iteration, we can consider a class of simulation-based asynchronous algorithms where we can alternate between simulations for policy improvement and simulations for policy evaluation.

Unfortunately, depending upon the relative ordering of the policy improvement and policy evaluation updates and the particular scheme used for the evaluation updates, this algorithm can diverge even for the simple case of having an exact value function representation and assuming that all states are updated infinitely often.

The one case that does have some convergence guarantees is referred to by Bertsekas and Tsitsiklis as *optimistic policy iteration* and always follows an evaluation step with a policy updating step. Recall, from Page 198 that for the non-simulation asynchronous policy iteration, following an evaluation step with a policy update step was equivalent to asynchronous value iteration. Thus, optimistic policy iteration is nearly the same as simulation-based asynchronous value iteration. However, if the evaluations are based on anything other than single transition samples (i.e., TD(λ) for $\lambda >> 0$), then convergence cannot generally be guaranteed.

Q-learning

In the POMDP RL/NDP algorithms presented later, we will only consider using approximations to the optimal Q-functions. Additionally, we will focus our attention on a form of asynchronous value iteration using the Q-function representation of the value function. The optimal Q-functions satisfy

$$V^{*,a}(s) = r(s,a) + \rho \sum_{s'} \tau(s,a,s') \max_{a'} V^{*,a'}(s') \quad , \tag{5.9}$$

and the normal value iteration algorithm can be rephrased in terms of Q-functions as

$$V^{a}(s) := r(s,a) + \rho \sum_{s'} \tau(s,a,s') \max_{a'} V^{a'}(s') \quad ,$$

where we must iterate over time, states and actions. Note that the normal VI algorithm also iterations over actions, only the intermediate Q-functions are not explicitly stored. The results of asynchronous value iteration hold for an asynchronous VI with Q-functions, so as long as each state-action pair is updated infinitely often, these values converge to the optimal values.

We can convert this to its small step-size version

$$V^a(s) := (1-\eta)V^a(s) + \eta\left(r(s,a) + \rho\sum_{s'}\tau(s,a,s')\max_{a'}V^{a'}(s')\right) \quad ,$$

and finally derive the single sample, Robbins-Monro stochastic approximation algorithm (based upon Equation 5.9), yielding

$$V^a(s) := (1-\eta)V^a(s) + \eta\left(r + \rho\max_{a'}V^{a'}(s')\right) \quad . \tag{5.10}$$

Here the particular sample transition is from state s to s' for action a where the immediate reward r is received. When the samples are generated from simulations, using Equation 5.10 results in exactly Watkins' Q-learning algorithm [126]. Using the convergence results of the stochastic approximation algorithms, we immediately get the convergence of Q-learning.

The experiments we will use for our RL/NDP POMDP algorithms are based precisely on simulation-based value iteration, using Q-functions to represent the value function. The only differences are that we use approximations to the Q-functions and we have a continuous state space. Recall, that this also requires the step-size parameter to satisfy the properties given in Equations 5.4 and 5.5.

In practice, without the ability to update every state-action pair infinitely often, the specifics of the sampling have a major impact on the empirical performance of the algorithm. In particular, the starting states and the action selection define the trajectories that will be taken. If certain states are not visited, or certain actions are not tried in some states, then advantageous actions and important parts of the state space will be missed. For this reason, the Q-learning algorithm often employs an exploration strategy, where the currently best action selection is supplemented with exploratory actions to ensure advantageous alternatives are not missed.

5.1.6 Simulation-based DP with Function Approximation

We now come to the last step in constructing the full RL/NDP framework: adding a function approximator. The simulation-based dynamic programming techniques of the previous section were presented in the context of exact value function representation and predominantly have pleasing theoretical convergence results. Additionally, we have seen that incremental gradient methods for solving a least squares optimization problem also have some pleasing convergence guarantees. Unfortunately, combining function approximation and simulation-based dynamic programming do not generally lead to pleasing convergence results. However, they are often combined in practice with impressive results [122, 32].

Additionally, though there is little theoretical basis for the simulation-based DP algorithm for the case of continuous state spaces, with function approximation, there is no inherent limitations on the state space size and we proceed here assuming we are in the POMDP realm of continuous state spaces.

We have seen the incremental gradient algorithm as a stochastic approximation algorithm:

$$\begin{aligned}\Gamma &:= (1-\eta)\Gamma + \eta\,(\Gamma - \nabla\overline{e}) \\ &:= \Gamma - \eta\nabla\frac{1}{2}e^2 \\ &:= \Gamma + \eta\nabla\widetilde{V}(\Gamma, b)e \quad ,\end{aligned} \tag{5.11}$$

where here we have removed the subscript m on e to show that we no longer consider having an explicit data set. Note that our RL/NDP approaches to POMDPs use this as the basis for their parameter update rule.

Where we previously defined $e = V^*(b) - \widetilde{V}(\Gamma, b)$, which is the true error, we can view (somewhat incorrectly) $r + \rho\widetilde{V}(\Gamma, b')$ as a sample of $V^*(b)$ based upon a transition from state b to state b' with reward r received. This is a commonly used update rule, and one we explore in our experiments, except it has a tenuous formal basis. However, it is closely related to an approach which does have a more formal basis.

Residual Gradient Method

The optimal value function for an MDP satisfies $V^* = TV^*$ and is in fact the only value function that satisfies this relationship: i.e., it is a unique fix-point of the operator T[1]. The *Bellman residual* at any point in the the value iteration algorithm is defined to be $TV - V$. Since the VI algorithm converges, the Bellman residual is guaranteed to be monotonically decreasing as the algorithm progresses, finally becoming zero when the optimal value function is achieved. Thus, an alternative view of attempting to find the optimal value function, is attempting to make the Bellman residual as small

[1]This is true for both COMDP and POMDP problems.

as possible.

With a function approximator, the current Bellman residual can be defined as

$$r + \rho \sum_{b'} \psi(b, a, b')\widetilde{V}(\Gamma, b') - \widetilde{V}(\Gamma, b) \ .$$

Thus, we can use a gradient descent algorithm to minimize this quantity by letting e be the Bellman residual and using Equation 5.11, which yields

$$\begin{aligned} \Gamma &:= \Gamma - \eta \nabla \frac{1}{2} e^2 \\ &= \Gamma - \eta \nabla \frac{1}{2} e^2 \\ &= \Gamma - \eta \nabla e e \\ &= \Gamma - \eta \nabla \left(r + \rho \sum_{b'} \psi(b, a, b')\widetilde{V}(\Gamma, b') - \widetilde{V}(\Gamma, b) \right) e \ . \end{aligned} \tag{5.12}$$

There is a problem using this within the RL/NDP framework since there is the explicit summations over states. We will adjust this to incorporate sampling below, but first discuss the implications of this iteration in relation to the proposed update scheme of Equation 5.11.

In the derivation of the gradient of the error term when we assumed we had a fixed data set with optimal output values, we made the following simplification:

$$\nabla \left(V^*(b) - \widetilde{V}(\Gamma, b) \right) = -\nabla \widetilde{V}(\Gamma, b) \ .$$

This same assumption is used in the algorithm based upon the update of Equation 5.11.

The assumption that the derivative of $V^*(b) = 0$ is correct when it represents the output portion of a training instance. However, for the RL/NDP methods based on the Bellman residual, when we replace $V^*(b)$ with the one-step value using the approximation for the next states, this assumption is no longer valid, since $\widetilde{V}(b')$ is also a function of the current parameters. This is easily seen in Equation 5.12 where the function approximator appears twice in the gradient term.

When the training output is the true value, the only way to reduce the error in the approximation is by adjusting the parameters in the direction of the training output, $V^*(b)$. When the Bellman residual is used in place of the training output, there are two ways to reduce the error; adjust the current estimate or adjust the current one-step look-ahead estimate.

Although many successful RL/NDP methods use the zero derivative assumption of Equation 5.11, called *direct* gradient methods by Baird, they have inferior theoretical convergence properties to the methods, called *residual gradient*, which do not make this assumption [5, 11]. There are simple examples that can be constructed where the direct method diverges.

Incorporation of the extra gradient term gives the residual gradient algorithm a more solid theoretical basis, but complicates the replacements of the explicit summations with samples. In the direct gradient version of Equation 5.11, we could use a single transition sample getting b, a, b' and r. The obvious, though incorrect, incorporation of a sample in Equation 5.12 would insert the sample in place of the two explicit summations and we would have

$$\Gamma = \Gamma - \eta \nabla \left(r + \rho \widetilde{V}(\Gamma, b') - \widetilde{V}(\Gamma, b) \right) \left(r + \rho \widetilde{V}(\Gamma, b') - \widetilde{V}(\Gamma, b) \right) \; .$$

In order to guarantee convergence of a stochastic iteration based upon

Equation 5.12 we need the expected value of the entire adjustment to be equal to the result of replacing the samples with their expected values; i.e., we require

$$\begin{aligned}&E\left[\nabla\left(r+\rho\widetilde{V}(\Gamma,b')-\widetilde{V}(\Gamma,b)\right)\left(r+\rho\widetilde{V}(\Gamma,b')-\widetilde{V}(\Gamma,b)\right)\right]\\&\quad=E\left[\nabla\left(r+\rho\widetilde{V}(\Gamma,b')-\widetilde{V}(\Gamma,b)\right)\right]E\left[\left(r+\rho\widetilde{V}(\Gamma,b')-\widetilde{V}(\Gamma,b)\right)\right] \quad .\end{aligned}$$

Because we need the expectation of the product of two random variables to be same as the product of the expectations, we need the random variables to be independent, which is not achieved by using the same sample in both places. This requires sampling the transition from state b twice. Thus, the correct update rule would be

$$\Gamma=\Gamma-\eta\nabla\left(r+\rho\widetilde{V}(\Gamma,b'')-\widetilde{V}(\Gamma,b)\right)\left(r+\rho\widetilde{V}(\Gamma,b')-\widetilde{V}(\Gamma,b)\right) \quad ,$$

where b' and b'' are two independently sampled next states for the initial state b.

This two-sample version is referred to as the *residual* gradient, and as mentioned has better theoretical convergence guarantees. However, this method is slow to converge, whereas the direct gradient, if it converges, converges faster [5]. Baird proposed using a weighting between the two methods to get the advantages of both methods. Because the residual and direct gradients are similar, this descent direction is given by

$$\Gamma:=\Gamma-\eta\left(\theta\nabla\rho\widetilde{V}(\Gamma_n,b'')-\nabla\widetilde{V}(\Gamma_n,b)\right)e \quad ,$$

where θ is the weighting factor.

5.2 Function Approximators for POMDPs

In this section we look at a number of available choices for representing an approximate value function over belief space for a POMDP. We give some background and derive the required parameter update rules for these function approximators.

The basic direct gradient update formula used is from Equation 5.11, which was

$$\Gamma := \Gamma + \eta \nabla \widetilde{V}(\Gamma, b) e \quad ,$$

where simulated experiences of transitioning from state b to state b' and receiving reward r would yield:

$$e = r + \rho \widetilde{V}(\Gamma, b') - \widetilde{V}(\Gamma, b) \quad .$$

5.2.1 Value vs. Q-functions

Regardless of the particular choice for a function approximator, there is also a choice of whether to use a single set of parameters, $\widetilde{V}(\Gamma, b)$, or a separate set of parameters for each action, $\widetilde{V}(\Gamma^a, b)$, which would be approximations to the Q-functions with

$$\widetilde{V}(\Gamma, b) = \max_a \widetilde{V}(\Gamma^a, b) \quad .$$

The main disadvantage of the single function approximator approach is that it requires an explicit model in order to convert the value function into a policy. Naturally, this is only a problem if the policy is not fixed and we would like to take the best action according to our current approximation. Although the experiments here have access to the full model, we would like to avoid doing an explicit summation over all possible next states and rely only on simulation steps to generate next states. By maintaining a separate function approximator for each action, a policy is readily available by evaluating each function approximator and taking the action that gives the best value.

There are two main problems with maintaining separate function approximators for each action. If the number of actions is large, then the space requirements could become prohibitive. A related problem is the speed of convergence. With a single function approximator, all simulated experiences contribute to adjusting a single value function, but with separate function approximators, only a percentage of the experiences go toward each function approximator. This means that it could require more simulated experiences to get reasonable Q-function approximations than would be required for the single $\widetilde{V}(\Gamma, b)$ approximator.

For updating approximations of Q-functions we incorporate the action taken, a, into our experience, so the direct gradient update rule becomes

$$\Gamma^a := \Gamma^a + \eta \nabla \widetilde{V}(\Gamma^a, b) e \quad ,$$

with error term

$$\begin{aligned} e &= r + \rho \widetilde{V}(\Gamma, b') - \widetilde{V}(\Gamma^a, b) \\ &= r + \rho \left(\max_{a'} \widetilde{V}(\Gamma^{a'}, b') \right) - \widetilde{V}(\Gamma^a, b) \quad . \end{aligned}$$

5.2.2 PWLC Representation

As for any learning algorithm, there is a need to have some bias in order to perform well. Given that we know the optimal infinite horizon value function is convex and possibly p.w.l. and convex, we can use this to bias the structure of the value functions. Here we consider two instances of a PWLC representation, one where we assume there is one and only one linear segment for each Q-function and one where multiple vectors are allowed.

Linear Q-functions

Using a single vector for each action we have a series of function approximators $\widetilde{V}^a(b) = \gamma^a \cdot b$, and $\widetilde{V}(\Gamma^a, b) = \max_a(b \cdot \gamma^a)$ with parameter set $\Gamma = \{\gamma^a | \forall a \in \mathcal{A}\}$. When a simulation step is taken and action a performed, the gradient, $\nabla\widetilde{V}^a(b)$ is comprised of the partial derivatives

$$\begin{aligned}
\frac{\partial \widetilde{V}^a(b)}{\partial \gamma_s^a} &= \frac{\partial}{\partial \gamma_s^a} \sum_{s'} b(s')\gamma_{s'}^a \\
&= \frac{\partial}{\partial \gamma_s^a} b(s)\gamma_s^a \\
&= b(s) \quad ,
\end{aligned}$$

where γ_s^a is the s^{th} component of vector γ^a.

Thus, the function approximator parameter adjustments for the individual components are

$$\gamma_s^a := \gamma_s^a + \Delta\gamma_s^a \quad , \tag{5.13}$$

where

$$\begin{aligned}
\Delta\gamma_s^a &= \eta \nabla \widetilde{V}^a(b) e \\
&= \eta b(s) \left(r + \rho \widetilde{V}(\Gamma, b') - \widetilde{V}^a(b) \right) \\
&= \eta b(s) \left(r + \rho \max_{a'} \left[b' \cdot \gamma^{a'} \right] - b \cdot \gamma^a(b) \right) \quad .
\end{aligned}$$

The update rule in Equation 5.13 is referred to as the linear-Q (LIN-Q) update rule for POMDPs. Work by Chrisman [28] in trying to simultaneously learn and act in POMDPs used a similar, though slightly incorrect, update rule. This is discussed more extensively in work by Littman, Cassandra and Kaelbling [68].

k-PWLC Representation

The natural extension to the linear Q-function representation is to allow a general PWLC representation with multiple vectors for each $\widetilde{V}^a(b)$. With more vectors in the PWLC approximation, the value function has the ability to closely represent a much wider range of value functions.

A PWLC value function is not smooth, which presents some difficulty, since this lack of smoothness transfers to the error function. Taking the derivative of a non-smooth function requires segmenting the function into pieces, each of which is smooth. For a function where there are fixed discontinuities, this is only a minor inconvenience, however, the places where this error function is discontinuous vary as the parameters of the value function changes.

One alternative is to ignore the dependence of the discontinuity on the parameters, and simply take the derivative of the function at the point of interest. For a PWLC value function, doing this results in a very simple update rule; it is essentially the LIN-Q update rule applied to the maximal vector in the representation of $\widetilde{V}^a(b)$.

The problem with this approach is the lack of any changes to vectors which are not maximal. Because of this, any vector which is dominated by all of the other vectors will never be updated, unless it becomes undominated from adjustments to the other vectors. We call these vectors *sunken* and this is especially problematic if the value function is initialized with a set of random vectors. Any vectors which start off dominated may never have their components updated, which wastes some of the representational power of this approach.

The best way to prevent sunken vectors is to make sure the initial value function consists of vectors that are not completely dominated by the others. However, this provides only minor relief, since there are still no guarantees that vectors will not become sunken. Nevertheless, we will use the approach in some of our empircal comparisons.

When initializing a k-PWLC function aproximator to random values, the specific initialization used when there are k vectors and N states breaks down into two cases: when $k \leq N$ and when $k > N$. For the former, we simply set $\lfloor N/k \rfloor$ unique components of each vector to have the maximal value in the random initial range. This ensures a non-empty region for each vector to start. For the case when $k > N$, we simply set a unique component for each of N vectors to the maximal value and allow the remaining $k - N$ vectors to have random ranges, possibly having them start out as sunken vectors. However, it does guarantee that at least N vectors start off with a non-empty region, though there are no guarantees that these vectors will not become sunken as the value function is adjusted.

5.2.3 The L_k Norm

We saw that extending the LIN-Q update rule for more than one vector per action was problematic, since gradient descent methods are best applied to continuous functions where the derivative can be taken. To combat this, the SPOVA algorithm of Parr and Russell [96] uses the L_k norm, which is a smooth approximation for the max operator.

The L_k norm is a continuous function with the nice property that the simple maximization of a regular PWLC function is its limiting case. If we have a set of vectors $\Gamma = \{\gamma^1, \gamma^2, \ldots, \gamma^L\}$, then we define the L_k norm value function for a belief state b with

$$V(\Gamma, b) = \sqrt[k]{\sum_{1 \leq l \leq L} (b \cdot \gamma^l)^k} \ .$$

To ensure the k^{th} root is not a complex number, this requires all of the γ^l components to be positive, which translates into having all positive rewards in the POMDP model, i.e., $\forall a, s, r(s, a) \geq 0$.

As $k \to \infty$, the L_k norm value function approaches

$$V(\Gamma, b) = \max_{\gamma^l \in \Gamma} b \cdot \gamma^l \ ,$$

which is simply a PWLC function.

In the SPOVA algorithm, this approximator is used to represent the $\widetilde{V}(\Gamma, b)$ function, though here we will explore using it using Q-functions. For our purposes, we will only consider $k > 1$.

The gradient of this representation[2] with respect to the γ^l_s parameters,

[2]There is a typographical error in the gradient formula as it appears in Parr and

letting $V(\Gamma, b)$ be $\widetilde{V}(\Gamma, b)$ or $\widetilde{V}^a(b)$, is

$$\begin{aligned}
\frac{\partial V(\Gamma, b)}{\partial \gamma_s^l} &= \frac{\partial}{\partial \gamma_s^l}\left(\sqrt[k]{\sum_{1\le j\le L} (b\cdot\gamma^j)^k}\right) \\
&= \frac{1}{k}\left(\sum_{1\le j\le L} \left(b\cdot\gamma^j\right)^k\right)^{\frac{1-k}{k}} \frac{\partial}{\partial \gamma_s^l} \sum_{1\le j\le L} \left(b\cdot\gamma^j\right)^k \\
&= \frac{1}{k}\left(\sum_{1\le j\le L} \left(b\cdot\gamma^j\right)^k\right)^{\frac{1-k}{k}} \sum_{1\le j\le L} \frac{\partial}{\partial \gamma_s^l} \left(b\cdot\gamma^j\right)^k \\
&= \frac{1}{k}\left(\sum_{1\le j\le L} \left(b\cdot\gamma^j\right)^k\right)^{\frac{1-k}{k}} \sum_{1\le j\le L} k\left(b\cdot\gamma^j\right)^{k-1} \frac{\partial}{\partial \gamma_s^l} b\cdot\gamma^j \\
&= \frac{1}{k}\left(\sum_{1\le j\le L} \left(b\cdot\gamma^j\right)^k\right)^{\frac{1-k}{k}} k\left(b\cdot\gamma^l\right)^{k-1} b(s) \\
&= \left(\sum_{1\le j\le L} \left(b\cdot\gamma^j\right)^k\right)^{\frac{1-k}{k}} \left(b\cdot\gamma^l\right)^{k-1} b(s) \\
&= \frac{b(s)\left(b\cdot\gamma^l\right)^{k-1}}{V(\Gamma, b)^{k-1}} \quad .
\end{aligned}$$

Note that as $k \to \infty$, the denominator approaches $(b\cdot\gamma^l)^{k-1}$ and the SPOVA update rule approaches the LIN-Q update rule. The two main difficulties with the SPOVA algorithm is in selecting and adjusting the exponent k and deciding the number of vectors, L, to use in the approximation.

The reported results for SPOVA [96] used a heuristic schedule for the exponent, starting k around 1.2 and increasing it linearly until it reached 8.0. A problem with adjusting the exponent is that changing the exponent

Russell's [96] original paper. The gradient formula in that paper appears with both the term in the numerator and denominator being raised to the k^{th} power instead of the $k-1^{\text{st}}$ power.

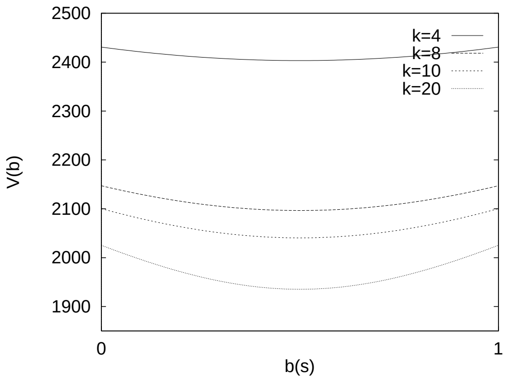


Figure 5.1: An L_k norm value function with varying values for the exponent k.

changes both the shape and range of the function. Although the change in shape is the desired property, making it more "PWLC", the change in the ranges of the value function could force the algorithm to need more training instances as it tries to recover the proper range of values. For instance, Figure 5.1 shows an L_k norm value function, using the identical set Γ, but varying the exponent k. If the changes are gradual enough, then the effects might not be so detrimental, but still a certain amount of the updating effort must be put into adjusting the ranges of the values as well as the general shape of the function.

Scaling after Adjusting the Exponent

A possible solution to this problem is to explicitly raise the value function to compensate for the adjustment of the exponent before more gradient updates are done. Suppose we have a heuristic exponent adjustment schedule that periodically raises the exponent by ϵ. Raising the exponent will lower the value function as well as changing its shape. If we suppose that the value function was in the right range, then we would like to take the value function with the new exponent, $k+\epsilon$, and raise it to be in the same range it previously was.

Since the value function's shape changes it become hard to define exactly how much and at which points the value function should be raised. However, a simple scheme is to simply try to make the new values match the old values at the simplex corners. This is just a specific instance of a more general scheme where we could try to make it match at any set of points. Below we derive the necessary change in the γ^l vectors for this instance.

Let e_s be the simplex corner corresponding to the information state where the entire probability mass is in state s. Prior to adjusting the exponent we have

$$V(\Gamma, e_s) = \left(\sum_{1 \le l \le L} \left(\gamma_s^l\right)^k \right)^{\frac{1}{k}} ,$$

where γ_s^l is the s^{th} component of the vector γ^l. Adjusting the exponent by ϵ yields a new value at the point e_s of

$$\widehat{V}(\Gamma, e_s) = \left(\sum_{1 \le l \le L} \left(\gamma_s^l\right)^{k+\epsilon} \right)^{\frac{1}{k+\epsilon}} .$$

We assume that whatever adjustment we want to make is distributed equally among the L vectors. Thus we are interested in finding δ such that

$$V(\Gamma, e_s) = \left(\sum_{1 \le l \le L} \left(\gamma_s^l + \delta\right)^{k+\epsilon} \right)^{\frac{1}{k+\epsilon}} .$$

This being hard to solve for δ directly, consider the simpler problem of trying to find $\overline{\delta}$, in

$$V(\Gamma, e_s) = \left[\sum_{1 \le l \le L} \left(\left(\gamma_s^l \right)^{k+\epsilon} + \overline{\delta} \right) \right]^{\frac{1}{k+\epsilon}} ,$$

which is simply assuming that the terms in the sum should all be adjusted by the same amount. From this we see that

$$\begin{aligned} V^{k+\epsilon}(\Gamma, e_s) &= \sum_{1 \le l \le L} \left(\gamma_s^l \right)^{k+\epsilon} + L\overline{\delta} \\ V^{k+\epsilon}(\Gamma, e_s) &= \widehat{V}^{k+\epsilon}(\Gamma, e_s) + L\overline{\delta} \\ \overline{\delta} &= \frac{V^{k+\epsilon}(\Gamma, e_s) - \widehat{V}^{k+\epsilon}(\Gamma, e_s)}{L} . \end{aligned}$$

Now we assume that the contribution of $\overline{\delta}$ quantity is equally distributed among all L vectors. This means we want each term to satisfy

$$\begin{aligned} \left(\gamma_s^l \right)^{k+\epsilon} + \overline{\delta} &= \left(\gamma_s^l + \delta \right)^{k+\epsilon} \\ \delta &= \left(\left(\gamma_s^l \right)^{k+\epsilon} + \overline{\delta} \right)^{\frac{1}{k+\epsilon}} - \gamma_s^l \\ \delta &= \left(\left(\gamma_s^l \right)^{k+\epsilon} + \frac{V^{k+\epsilon}(\Gamma, e_s) - \widehat{V}^{k+\epsilon}(\Gamma, e_s)}{L} \right)^{\frac{1}{k+\epsilon}} - \gamma_s^l . \end{aligned}$$

Therefore, the procedure for scaling the value function when adjusting the exponent by ϵ is:

1. compute the current value function, $V(\Gamma, e_s)$, at all the simplex corners e_s;

2. adjust the exponent by ϵ;

3. compute the new value function, $\widehat{V}(\Gamma, e_s)$ at all the simplex corners e_s;

4. for all $|\mathcal{S}|$ component of all L vectors set their new value with the assignment
$$\gamma_s^l := \left(\left(\gamma_s^l\right)^{k+\epsilon} + \frac{V^{k+\epsilon}(\Gamma, e_s) - \widehat{V}^{k+\epsilon}(\Gamma, e_s)}{L} \right)^{\frac{1}{k+\epsilon}} .$$

Gradient Descent on the Exponent

An alternative approach is to view the exponent as just another parameter of the function approximator, making the approximator $\widetilde{V}(\Gamma, k, b)$, and use a gradient update rule for it. Then, instead of updating just the vectors, the exponent is also adjusted.

To derive the partial derivative of $V(\Gamma, k, b)$ with respect to k, we note that
$$\frac{\partial g(x)^{h(x)}}{\partial x} = g(x)^{h(x)} \left[\frac{h(x)}{g(x)} \frac{\partial g(x)}{\partial x} + \frac{\partial h(x)}{\partial x} \ln g(x) \right] , \tag{5.14}$$
where for our purposes $x = k$, $g(x) = \sum_{1 \le l \le L} \left(b \cdot \gamma^l\right)^k = V(\Gamma, k, b)^k$ and $h(x) = 1/k$.

$$\begin{aligned}
\frac{\partial V(\Gamma, k, b)}{\partial k} &= V(\Gamma, k, b) \left[\frac{1}{kV(\Gamma, k, b)^k} \left(\frac{\partial}{\partial k} \sum_{1 \le l \le L} \left(b \cdot \gamma^l\right)^k \right) \right. \\
&\qquad \left. + \left(\frac{\partial}{\partial k} \frac{1}{k} \right) \ln V(\Gamma, k, b)^k \right] \\
&= V(\Gamma, k, b) \left[\frac{1}{kV(\Gamma, k, b)^k} \sum_{1 \le l \le L} \left(\frac{\partial}{\partial k} \left(b \cdot \gamma^l\right)^k \right) \right. \\
&\qquad \left. - \frac{1}{k^2} \ln V(\Gamma, k, b)^k \right] \\
&= V(\Gamma, k, b) \left[\frac{1}{kV(\Gamma, k, b)^k} \sum_{1 \le l \le L} \left(\frac{\partial}{\partial k} \left(b \cdot \gamma^l\right)^k \right) \right. \\
&\qquad \left. - \frac{1}{k} \ln V(\Gamma, k, b) \right] .
\end{aligned}$$

Then again using the identity from Equation 5.14 where $h(x) = k$ and $g(x) = b \cdot \gamma^l$, we have

$$\frac{\partial}{\partial k}\left(b \cdot \gamma^l\right)^k = \left(b \cdot \gamma^l\right)^k \ln\left(b \cdot \gamma^l\right) \quad ,$$

which makes the entire partial derivative

$$\begin{aligned}\frac{\partial V(\Gamma, k, b)}{\partial k} = \frac{V(\Gamma, k, b)}{k}\Bigg[&\frac{1}{V(\Gamma, k, b)^k}\sum_{1 \leq l \leq L}\left(b \cdot \gamma^l\right)^k \ln\left(b \cdot \gamma^l\right) \\ &- \ln V(\Gamma, k, b)\Bigg]\end{aligned} \tag{5.15}$$

Another complication that arises in considering adjusting the L_k norm's exponent through a gradient descent rule is the requirement of a separate learning rate for the exponent. Although using the same learning rate as for adjusting the γ^l components is possible, the nature of the differences between the two quantities suggests that best results will be achieved with separate learning rates.

5.3 RL/NDP Empirical Results

This section presents empirical results using some of the POMDP function approximators on a range of problems. The vast number of options and parameters available in the RL/NDP framework precluded trying all combinations and also complicates the interpretation of the resulting data. Our main focus here is comparing the function approximators with some minor exploration into comparing some of the parameters. The next sub-section discusses the basic structure of the experiments.

5.3.1 Experimental Set-up

For all the function approximators considered here, we explore only using Q-functions. Although this approach is not desirable for problems with a large set of actions, the problems considered here all have a relatively small action set.

We use the incremental gradient method with the update rules shown in Section 5.2. This can be viewed as an optimistic policy iteration algorithm using TD(0) value updates. A training instance consists of some number of trajectories, either $1,000$ or $10,000$, truncated to 100 steps, which results in $100,000$ or $1,000,000$ specific training steps. However, problems where there are zero-cost absorbing would normally result in fewer steps, since the full 100 steps per trajectory may not be reached. For the absorbing state problems, the minimum number of additional trajectories were used to bring the number of training steps up to at least $100,000$ or $1,000,000$ steps. Thus, for those problems there could be up to 99 more training steps than the non-absorbing state problems.

Step Size	Training Steps Interval
0.1	$[\,0 \; 25,000\,]$
0.01	$[\,25,001 \; 50,000\,]$
0.001	$[\,50,001 \; 75,000\,]$
0.0001	$[\,75,001 \; \infty\,)$

Table 5.2: Step-size adjustment schedule for 100, 000 training step RL/NDP experiments.

Regardless of the number of trajectories used, the parameter training phase is followed by an evaluation of the resulting value function using 10, 000 trajectories of 100 steps and averaging the discounted reward received for each trajectory. Naturally, no exploratory actions are taken in the evaluation phase. Also, no additional trajectories for the zero-cost absorbing state problems are needed in the evaluation phase, since each trajectory is a single sample of the discounted reward, no matter how many actual steps are in the trajectory.

For each trajectory, the starting state is chosen to be consistent with a problem-specific initial information state. The initial information state reflects some prior probabilities on the state for the various problems.

The basic structure of the algorithm follows the Q-learning approach, where the best action according to the current Q-functions is usually taken, but with probability 0.25 a random exploratory action is used. Additionally, the step-size increment, or *learning rate* has the general form of being decreased over time and the specific schedule used for the 100, 000 training step experiments is given in Table 5.2 with Table 5.3 giving the schedule for the 1, 000, 000 step experiments.

Because the k-PWLC algorithm has more vectors and only one vector is

Step Size	Training Steps Interval
0.1	[0 250, 000]
0.01	[250, 001 500, 000]
0.001	[500, 001 750, 000]
0.0001	[750, 001 ∞)

Table 5.3: Step-size adjustment schedule for 1, 000, 000 training step RL/NDP experiments.

Step Size	Training Steps Interval
0.1	[0 75, 000]
0.01	[75, 001 150, 000]
0.001	[150, 001 225, 000]
0.0001	[225, 001 ∞)

Table 5.4: Step-size adjustment schedule for 300, 000 training step 3-PWLC experiments.

updated per step, the amount of experience that goes into adjusting each vector can be $1/k$-th as much as the LIN-Q algorithm. To compensate, we ran the k-PWLC algorithms with k more training steps than their LIN-Q counterparts. However, this may result in some single vector being updated more than would be in the LIN-Q experiments. These longer training phases necessitate an adjusted step-size schedule, and Tables 5.4 through 5.7 show the schedule used for the k-PWLC experiments.

Step Size	Training Steps Interval
0.1	[0 750, 000]
0.01	[750, 001 1, 500, 000]
0.001	[1, 500, 001 2, 250, 000]
0.0001	[2, 250, 001 ∞)

Table 5.5: Step-size adjustment schedule for 3, 000, 000 training step 3-PWLC experiments.

Step Size	**Training Steps Interval**
0.1	$[\,0\ 175,000\,]$
0.01	$[\,175,001\ 350,000\,]$
0.001	$[\,350,001\ 525,000\,]$
0.0001	$[\,525,001\ \infty\,)$

Table 5.6: Step-size adjustment schedule for 700, 000 training step 7-PWLC experiments.

Step Size	**Training Steps Interval**
0.1	$[\,0\ 1,750,000\,]$
0.01	$[\,1,750,001\ 3,500,000\,]$
0.001	$[\,3,500,001\ 5,250,000\,]$
0.0001	$[\,5,250,001\ \infty\,)$

Table 5.7: Step-size adjustment schedule for 7, 000, 000 training step 7-PWLC experiments.

Since there often is randomness[3] in the initial value functions and always randomness in the simulated trajectories, performing the training and evaluation phase will yield different results from one instance to another. To alleviate this, we repeated each experiment 10 times, where an experiment consists of a certain number of training steps and a 10, 000 trajectory evaluation phase as discussed previously. This set-up introduces some complications in doing a statistical analysis on the data since there are now two sources of randomness contributing to the evaluation: the randomness in the training and the randomness in the evaluation. We have pooled all $10 \times 10,000$ evaluation trajectories into a single batch of 100, 000 samples and compared the algorithms on the basis of these batches. While this is not precisely correct, it does provide some insight into the performance of these algorithms.

[3] It is not random when the Q-functions are used for initialization.

5.3.2 Small Problems

In this section we explore the LIN-Q and k-PWLC schemes on the same set of small problems used in Section 4.9.2 and shown in Table 4.4. The purpose of using a set of small examples is to have some domains where we can compare these RL/NDP approaches to the optimal answers. For the k-PWLC algorithm we used the initialization technique discussed in Section 5.2.2.

The true infinite horizon optimal values for these small problems are actually larger than are shown. Since we used truncated trajectories, we are imposing a limit on the total discounted reward that can accumulate. For this reason, instead of calculating the optimal values, we evaluated the optimal controller using the same simulation set-up as the other algorithms. Because there is no randomness in producing the optimal answer, we only required evaluating the optimal answer once. Because the optimal answer reported is based upon simulation, the randomness of the simulation does not preclude another algorithm from performing better on a particular problem instance. The initial value function vectors were chosen to have random vectors where their components were randomly set to values in the interval $[-20, +20]$.

Table 5.8 shows the results for these small problems where the best entry for a problem is boxed. The lighter boxed entries indicate that these are not significantly worse than the best answer, where significance was determined by a simple two-sample T-test with $p = 0.995$.

As can be seen from this table, even for the $100,000$ training step experiments, the LIN-Q algorithm does exceedingly well. On some of the problems,

Alg.	**Steps**	4x3	4x4	CHEESE	PAINT	SHUTTLE	TIGER	NETWORK	NONLIN	SACI
LIN-Q	1×10^5	$\boxed{1.860}$	3.542	$\boxed{3.464}$	$\boxed{3.267}$	$\boxed{32.657}$	$\boxed{19.222}$	288.465	7.072	13.961
	1×10^6	$\boxed{1.868}$	$\boxed{3.708}$	$\boxed{3.465}$	$\boxed{3.268}$	$\boxed{32.690}$	$\boxed{19.285}$	$\boxed{290.040}$	$\boxed{7.158}$	$\boxed{14.787}$
3-PWLC	3×10^5	$\boxed{1.866}$	$\boxed{3.709}$	$\boxed{3.464}$	$\boxed{3.270}$	$\boxed{32.678}$	$\boxed{19.277}$	287.328	$\boxed{7.158}$	13.409
	3×10^6	1.802	$\boxed{3.709}$	3.398	3.213	$\boxed{32.663}$	$\boxed{19.254}$	$\boxed{291.343}$	$\boxed{7.158}$	$\boxed{14.762}$
7-PWLC	7×10^5	$\boxed{1.861}$	$\boxed{3.709}$	$\boxed{3.463}$	$\boxed{3.271}$	$\boxed{32.655}$	$\boxed{19.307}$	$\boxed{289.585}$	$\boxed{7.158}$	14.555
	7×10^6	1.832	$\boxed{3.710}$	3.388	3.181	$\boxed{32.656}$	$\boxed{19.261}$	$\boxed{290.933}$	$\boxed{7.158}$	$\boxed{14.760}$
Optimal	N/A	N/A	3.712	3.464	3.279	32.700	19.181	290.998	7.158	N/A

Table 5.8: LIN-Q and k-PWLC comparison on the suite of small problems using various numbers of training steps. Initial vector range $[\,-20\ \ +20\,]$. (mean). T-test with $p = 0.995$.

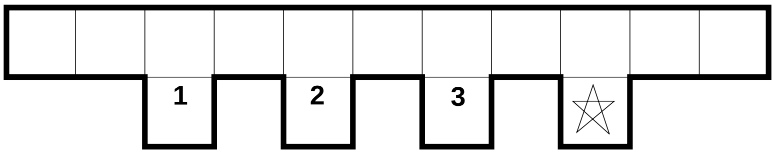


Figure 5.2: HALLWAY domain, a 57 state robot navigation domain.

training LIN-Q for longer improves the solution to near optimal behavior for those where the optimal controller could be computed. Since there is no room for improvement, any advantage by using multiple vectors, the 3-PWLC and 7-PWLC versions, would not be brought out by these problems.

5.3.3 Larger Problems

The results from the small problems establishes the potential for using these RL/NDP techniques, and the next step is an attempt to apply them to problems which are larger and which cannot be exactly solved. We first applied these algorithms to the two robot navigation domains shown in Figures 5.2 and 5.3 which have 57 and 89 states respectively. These domains are described in more detail elsewhere [68], but are similar to the navigation domains described in Appendix H.5. The starred locations are the goal locations which yields a +1 reward and resets the state to a random non-goal state. For both of these problems, the initial state is equally likely to be any of the non-goal locations and the discount factor used is 0.95.

We first ran LIN-Q and used the same initial random range of $[-20, +20]$ with the same training set-up of $100,000$ steps and evaluation with $10,000$ trajectories of length 100. The first difficulty we encounter is in gauging the performance of the resulting policies, since the optimal answer is not

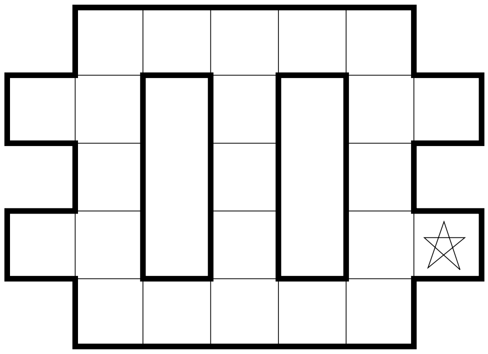


Figure 5.3: HALLWAY-2 domain, a 89 state robot navigation domain.

known. To this end, we ran an *omniscient* control strategy (OMNI) which can peek at the underlying state and perform the optimal action for the true underlying state, where the optimal action is computed by solving for the problem as if it was completely observable. Note that this omniscient controller can perform much better than the optimal partially observable control strategy, but does provide an upper bound on the performance that is achievable. In addition, we have included two other results to help gauge the performance.

- **Heuristic** - this is the performance of the best heuristic solution, which are to be discussed in the next chapter. However, we have hand-picked the best heuristic, and there is no current motivated way to know a priori which heuristic should be chosen for a given problem. Statistical comparisons between the best heuristic and these RL/NDP schemes can be found in Section 6.8 following the empirical results for the heuristics.

- **Human** - we developed a graphical simulation environment of these

Alg.	**Initial**	**Steps**	HALLWAY	HALLWAY-2
LIN-Q	$[-20\ +20]$	1×10^5	0.059	0.033
	$[-20\ +20]$	1×10^6	0.506	0.060
Heuristic	—	—	0.823	0.378
Human	—	—	0.865	0.300
OMNI	—	—	1.519	1.189

Table 5.9: LIN-Q on larger domains with random initialization.

domains which display the information state probabilities as varying shades of grey. A human[4] used this to select actions.

The results are shown in Table 5.9 where we see that there is still much room for improvement. The $100,000$ step experiments result in very poor performance, though more training steps helps to improve the solution. It may be that even longer training runs would continue to improve the results, but somewhat discouraging to think that more than $1,000,000$ training steps are required.

As the table shows, the quality of the answers is far from what is achievable by using simple heuristics. There are a few problems that contribute to this.

- With random initial vectors, early stages of the training are wasted as the actions cause very undirected trajectories that only occasionally lead to the goal and a positive reinforcement.

- The optimal answers values are in a much narrower range than the initial random value range, so it requires extensive training to move the values into the correct range.

[4]Thanks to Michael Littman for his patience in performing this task.

- With a larger state space than the small problems of the previous section, additional training trajectories are needed as a larger state space is explored.

The next few sub-sections address these issues.

5.3.4 Biasing the Training

In and attempt to assess the items listed above, we used a more motivated initialization scheme for the individual vectors in the LIN-Q Q-functions. As with any machine learning task, bias plays an important role in the quality of the solution, and we do not expect our task to be any different.

Recall from Equation 2.9 in Section 2.2.3 that a COMDP has associated value functions for each action called Q-functions. Since there is an underlying COMDP in a POMDP we would expect the optimal COMDP Q-functions to be somewhere in the correct value range for the optimal values of the POMDP. This makes the Q-functions a disciplined way to initializes the LIN-Q vectors which gives an approximate range on the values and potentially a way to seed the vectors with a reasonable initial policy.

The natural question to answer is how well a control job the Q-functions themselves would do without any adjustments. This control strategy is discussed about in more detail in Section 6.3 of the next chapter on heuristics under the name Q-MDP. We defer the details and discussion of this control heuristic to the next chapter, but will include the results from this heuristic to gauge whether the LIN-Q updating of these vectors is doing any useful work.

Table 5.10 shows the results of using the Q-functions to initialize the

Alg.	Initial	Steps	HALLWAY	HALLWAY-2
Q-MDP	—	—	0.344	0.097
LIN-Q	[−20 +20]	1×10^5	0.059	0.033
	[−20 +20]	1×10^6	0.506	0.060
	Q-func	1×10^5	0.910	0.218
	Q-func.	1×10^6	0.946	0.468
Heuristic	—	—	0.823	0.378
Human	—	—	0.865	0.300
OMNI	—	—	1.519	1.189

Table 5.10: LIN-Q on larger domains comparing random initialization and Q-functions.

LIN-Q vectors and compares it to the optimal, best heuristic and the Q-MDP control heuristic. As shown, in both domains the LIN-Q algorithm significantly improves upon the initial vectors and with enough training steps, surpasses both the human and best heuristic performance.

Using the Q-function for initialization yields very good performance, but it does not tell us if the benefits come from having a good early control strategy in the training or by starting the initial values in a more reasonable range. In an attempt to pry these two issues apart we ran the same experiments again using random initial vectors, but restricting the range of the value to those in the range of the Q-function values, which is [1.0 2.5].

Table 5.11 shows these results along with the previous results and we see that simply restricting the range helps a significant amount, but that it is not the sole contributor to the increase in performance. Although with enough training experience we can get high quality solutions starting with the restricted-range random vectors, using the Q-functions is quite helpful in reducing the number of training steps required and as a motivated method

Alg.	**Initial**	**Steps**	HALLWAY	HALLWAY-2
Q-MDP	—	—	0.344	0.097
LIN-Q	[−20 + 20]	1×10^5	0.059	0.033
	[−20 + 20]	1×10^6	0.506	0.060
	Q-func	1×10^5	0.910	0.218
	Q-func.	1×10^6	0.946	0.468
	[1.0 2.5]	1×10^5	0.755	0.099
	[1.0 2.5]	1×10^6	0.944	0.422
Heuristic	—	—	0.823	0.378
Human	—	—	0.865	0.300
OMNI	—	—	1.519	1.189

Table 5.11: LIN-Q on larger domains comparing various initial vector values.

for choosing the a useful initial range.

Finally, we want to see how the the potentially more expressive k-PWLC representations do on these domains, both to see if they are at all useful and also to see whether or not they can result in improved performance over LIN-Q. The full results on these two domains are shown in Table 5.12.

Here we see that not only are the k-PWLC representations useful, but they give significantly better results than those attained with the LIN-Q representation. Again, the dark boxes highlight the best value and lighter boxes show entries which are not significantly different as determined with a two-sample T-test with $p = 0.995$. Curiously though, the best results are obtained using the restricted-range random vector initialization and not the Q-functions. Although we have not explored why this is the case, we speculate that the Q-functions could be forcing the function toward some inferior local minimum in the error space, where the random vectors tend to be located near a better minimum.

Alg.	Initial	Steps	HALLWAY	HALLWAY-2
Q-MDP	—	—	0.344	0.097
LIN-Q	[−20 +20]	1×10^5	0.059	0.033
	[−20 +20]	1×10^6	0.506	0.060
	Q-func	1×10^5	0.910	0.218
	Q-func.	1×10^6	0.946	0.468
	[1.0 2.5]	1×10^5	0.755	0.099
	[1.0 2.5]	1×10^6	0.944	0.422
3-PWLC	[−20 +20]	3×10^5	0.031	0.042
	[−20 +20]	3×10^6	0.618	0.149
	Q-func	3×10^5	0.944	0.414
	Q-func.	3×10^6	0.946	0.477
	[1.0 2.5]	3×10^5	0.956	0.172
	[1.0 2.5]	3×10^6	$\boxed{1.007}$	0.501
7-PWLC	[−20 +20]	7×10^5	0.024	0.053
	[−20 +20]	7×10^6	0.717	0.174
	Q-func	7×10^5	0.942	0.466
	Q-func.	7×10^6	0.951	0.481
	[1.0 2.5]	7×10^5	0.942	0.466
	[1.0 2.5]	7×10^6	$\boxed{1.008}$	$\boxed{0.510}$
Heuristic	—	—	0.823	0.378
Human	—	—	0.865	0.300
OMNI	—	—	1.519	1.189

Table 5.12: LIN-Q and k-PWLC comparisons on 57 and 89 state POMDP problems using various initializations and number of training steps. T-test with $p = 0.995$

5.3.5 Other Domains

The small domains and the HALLWAY and HALLWAY-2 domains show the potential for the LIN-Q and k-PWLC algorithms, but are still a very limited class of problems, which may have special structure, allowing these algorithms to do particularly well. In an effort to explore these algorithms on a broader class of larger problems we have implemented simulators for a variety of domains. We first give a brief overview of these domains and then give the empirical results.

Domain Descriptions

The domains used fall into 5 classes:

- robot navigation domains (Appendix H.5) which include CIT, MIT, SUNYSB, PENTAGON, FOURTH;
- single aircraft identification or IFF (Appendix H.4);
- large baseball domain or BASEBALL (Appendix H.1);
- machine maintenance or MACHINE (Appendix H.3);
- slotted aloha network protocol (Appendix H.2) including ALOHA-10 and ALOHA-30;

The details of the domains are given in the appendices indicated above, but the overall problem sizes are given in Table 5.13. For all these domains a discount factor of 0.99 was used.

For the 5 robot navigation problems, the transition and observation probabilities are derived from the probabilities shown in Appendix H.5 with the

Name	States	Actions	Obs.
CIT	281	4	28
MIT	201	4	28
SUNYSB	297	4	28
PENTAGON	209	4	28
FOURTH	$1,049$	4	28
IFF	104	4	22
BASEBALL	$7,681$	6	9
MACHINE	256	4	16
ALOHA-10	30	9	3
ALOHA-30	90	29	3

Table 5.13: Various POMDP problem names and sizes.

Tables H.10 and H.11. The layout of these domains is identical to those shown in Figures 6.1 (CIT), 6.2 (MIT), 6.3 (SUNSB), 6.4 (PENTAGON) and 6.5 (FOURTH) of the next chapter, with a single starting state and a single goal state. Although these navigation problems have potentially 65 observations, for these 5 domains the `undetermined` observation has zero probability from all states, making these effectively 28 observation problems.

For many of the domains, the problem was scalable along a particular dimension.

- The IFF problem discretized the distances of the approaching aircraft into 10 locations.
- The BASEBALL problem is adjustable by the number of innings, though we used only a single inning game.
- The MACHINE problem, the problem is adjustable by the number of internal components for the machine. We used 4 for our example,

which results in 256 states, since each component has four possible states of condition it can be in.

- The ALOHA problems, the problem is adjustable by the maximal number of back-logged packets allowed. We used the values 10 and 30 for the ALOHA-10 and ALOHA-30 domains respectively.

Empirical Results

This larger suite of problems was run with initialization of both random vectors and the problems' corresponding Q-functions. We also ran both $100,000$ and $1,000,000$ step variations where the number of steps for the k-PWLC schemes were adjusted accordingly. The range of values used for the random initialization was $[\ -20\ \ +20\ ]$ for all but the 5 robot navigation problems, where here the components of the vectors were set randomly within the range $[\ 0.5\ 1.0\ ]$.

Tables 5.15 and 5.14 show the complete results for all the variations tried, the results from the best heuristic of the next chapter, as well as the omniscient and Q-MDP heuristics. A two sample T-test with $p = 0.995$ was used and is the basis for the boxed entries in the table; all boxed entries are not significantly different than the best entry.

Focusing on the robot navigation problems of Table 5.14 first, we see a strong sensitivity of the RL/NDP algorithms to the initial vectors used. In particular, for the $100,000$ step experiments the LIN-Q representation does miserably unless the Q-functions are used. We see that for the most part, a the LIN-Q representation suffices for doing well. The one exception, MIT, is a diabolically symmetric environment and the 7-PWLC representation does

Alg.	Init.	Steps	CIT	MIT	SUNYSB	PENT.	FOURTH
Q-MDP	—	—	0.832	0.812	0.759	0.821	0.590
LIN-Q	Rand.	1×10^5	-0.012	0.081	-0.026	0.060	-0.025
	Rand.	1×10^6	0.821	0.834	0.610	0.803	-0.008
	Q-func.	1×10^5	0.827	0.841	$\boxed{0.771}$	$\boxed{0.819}$	0.593
	Q-func.	1×10^6	$\boxed{0.828}$	0.846	0.755	0.818	$\boxed{0.596}$
3-PWLC	Rand.	3×10^5	0.821	0.830	0.273	0.790	-0.002
	Rand.	3×10^6	$\boxed{0.828}$	0.854	0.753	0.800	0.071
	Q-func.	3×10^5	$\boxed{0.828}$	0.848	0.756	0.815	$\boxed{0.595}$
	Q-func.	3×10^6	$\boxed{0.829}$	0.850	0.763	0.814	0.588
7-PWLC	Rand.	7×10^5	0.813	0.853	0.711	0.790	-0.012
	Rand.	7×10^6	0.825	$\boxed{0.868}$	0.757	0.805	0.557
	Q-func.	7×10^5	$\boxed{0.828}$	0.848	0.758	0.815	0.594
	Q-func.	7×10^6	0.827	0.857	0.766	0.814	0.591
Heur.	—	—	0.834	0.863	0.764	0.822	0.592
OMNI	—	—	0.845	0.894	0.809	0.836	0.625

Table 5.14: The LIN-Q and k-PWLC algorithms on some robot navigation problems. T-test with $p = 0.995$.

significantly better than the rest, though curiously, by starting with random initial vectors. Again, similar to the HALLWAY and HALLWAY-2 problems, this could be attributable to local minima.

Although the k-PWLC variations do well on some domains with random initialization, the larger and more complex domains also seem to require Q-function initialization for good performance. Increasing the amount of training also helps when starting from random vectors, but especially on the larger FOURTH domain, initializing with the Q-functions is required for good performance. In fact, only for the MIT and SUNYSB domains do these RL/NDP schemes provide significant improvement over the Q-MDP method. Based upon the omniscient controller, we see that there was little room for improvement on these domains anyway, though it is encouraging to know that these RL/NDP algorithms preserve the high quality performance of the Q-functions.

For the other domains in Table 5.15, we have some mixed results. For the IFF domain, The $1,000,000$ step 7-PWLC with random initial vectors provides the best performance. For the BASEBALL domain, a 3-PWLC variation is best, and for the remainder, the simpler LIN-Q representation does best. However, notice that in all of these domains the RL/NDP techniques yield fairly reasonable results.

Alg.	Init.	Steps	IFF	BB	MACH.	ALOHA10	ALOHA30
Q-MDP	—	—	4.496	0.101	59.693	127.429	851.035
LIN-Q	Rand.	1×10^5	8.136	0.217	13.463	75.128	666.820
	Rand.	1×10^6	6.971	0.096	4.500	73.004	602.963
	Q-func.	1×10^5	8.010	0.043	$\boxed{59.839}$	121.474	$\boxed{825.365}$
	Q-func.	1×10^6	8.504	0.063	58.230	$\boxed{123.871}$	811.769
3-PWLC	Rand.	3×10^5	7.875	0.196	6.328	69.511	628.338
	Rand.	3×10^6	8.630	$\boxed{0.481}$	7.913	70.139	630.400
	Q-func.	3×10^5	8.247	0.072	59.028	121.751	808.669
	Q-func.	3×10^6	8.441	0.265	50.954	111.760	581.529
7-PWLC	Rand.	7×10^5	8.361	0.464	3.299	70.350	642.026
	Rand.	7×10^6	$\boxed{8.828}$	N/A	8.053	69.293	659.705
	Q-func.	7×10^5	8.045	0.067	57.022	119.510	799.756
	Q-func.	7×10^6	8.218	0.328	42.972	119.316	626.691
Heur.	—	—	8.389	0.668	59.693	127.429	852.773
OMNI	—	—	10.079	0.658	66.236	145.572	937.143

Table 5.15: The LIN-Q and k-PWLC algorithms on the suite of larger problems. (mean) T-test with $p = 0.995$.

5.4 Related Work

The majority of work in reinforcement learning has used COMDPs as its basis, since the theory is better developed and the mathematical foundation more solid. However, there has always been interest in attacking the problem of partial observability. An early attempt to deal with partial observability was by Whitehead and Ballard [134], but it is only effective when the partial observability takes a special form, since it attempts to avoid the states which appear confusing.

The work by Lin and Mitchell [67] used recurrent neural networks to cope with partial observability. Knowing that good policies is these domains will require some type of memory, they present three architectures for maintaining this memory. In one architecture, they simply give the recurrent network the current action and observations, hoping the memory of the network will capture the needed structure. In another instance, a finite history of the process is given to the recurrent network. Their last architecture uses an indirect method, which has one network learning a model and providing the current state estimate to another network which attempts to learn the value function. Similar to the choice we had for our function approximators, they discuss the issue of using a single monolithic network or a single network for each action. They provide detailed comparisons on these different architectures and the structures of problems for which each may do best. However, all the problems presented there are relatively small, and no attempt is made to exploit properties of the optimal value function. It would be interesting to explore adding some bias on the value function to these connectionist schemes, since there have been a number of impressive application using neural network function approximators [122, 32].

Schmidhuber [110] has also looked at applying recurrent neural networks to deal with the problem of hidden state. Some later work by Wiering and Schmidhuber [135] deals with the non-Markovian nature of the POMDP control problem by breaking it down into a sequence of Markovian tasks. This greatly restricts the type of policies that are considered and requires some initial knowledge about how many tasks might be needed.

Chrisman [28] presented an indirect method where a predictive model of the POMDP is maintained and updated based upon experience. Here the predictive model is in the form of a *hidden Markov model*, rather than a recurrent network, and there is a rule to add states to the model when it yields poor predictions. Like the LIN-Q algorithm, the value function consists of a single vector per action and the update rule is similar to LIN-Q, though not identical. The differences are discussed, highlighted and empirically compared in work by Littman, Cassandra and Kaelbling [68].

Ring [104] combines the use of a recurrent neural network for the predictive model with rules for adjusting the model when a richer representation is needed. This is more of a hierarchical approach and allows handling of hidden state with a varying amout of representational complexity, depending upon the need for additional bits of information.

McCallum has worked extensively on applying RL to problems with hidden state [85, 86, 83, 84]. The end result of his efforts is a finite memory approach, where the amount of history required to make a decision can vary. The idea is to only add more history information if it will increase the utility of the policy. This is similar to Ring's work, though McCallum uses a tree-based representation instead of recurrent neural networks.

Some preliminary research in applying Q-learning directly to POMDPs has been undertaken by D'Ambrosio and Fung [33] using a table-based function approximator which maintains an entry for each belief state visited.

Crites [32] has successfully applied RL to the problem of elevator control using teams of reinforcement learning agents. Although not explicitly handling partial observability, the elevator control domain does have elements of hidden state; e.g., the actual number of persons waiting to board an elevator. Despite ignoring this, his system performs quite well in the face of partial observability. Additionally, he does some experiments varying the amount of partial observability and sees that his system is fairly robust to this. However, the form of the partial observability explored is heavily domain dependent and it is hard to say whether his techniques would be equally robust toward other forms of partial observability or in other domains. However, this successful use of teams of RL agents holds promise for dealing with partial observability directly.

There is alos some work in dealing with reinforcement learning in continuous spaces [93] and future advances in continuous state space RL algorithms would have direct applicability to POMDP problems.

5.5 Conclusions

In this chapter we have overviewed one general scheme for reinforcement learning (or neuro-dynamic programming) and then presented some instances for POMDPs that exploit knowledge of the shape of the value function.

We have shown that these technique do improve solutions and overall, combined with the work of others, suggest that RL/NDP techniques have a great potential as a basis for approximate algorithms for solving large POMDPs. There is still much research to be done in this area, especially in combining RL/NDP with feature-based approaches.

Chapter 6

Heuristic Approximations

Although the RL/NDPframework has a mathematical basis and some nice underlying theory, it can be a significant amount of machinery to wield, it could require extensive training and it is not a trivial task to get the right set of parameters. The natural question arises as to whether or not simpler, though perhaps less mathematically motivated, techniques could be employed to choose actions in the face of uncertainty. In this section we explore some simpler control rules which require no training of a value function, but of which little can be said theoretically. We empirically explore these methods on a range of domains. We note that there are intriguing, though unexplored, possibilities for combining these heuristics with an RL/NDP approach, where the heuristics are treated as features of the environment [25].

For all of the methods discussed, we are assuming that we can model the domain as a POMDP and that an explicit information state can be maintained at each step. Most of these heuristics first appeared in research by Cassandra, Kaelbling and Kurien [21].

There is a class of techniques in control theory called *certainty equivalent* *controllers* (CEC) which are closely related to the approach of some of these heuristics [9]. The controllers make the assumption that the state transition functions are deterministic, and control proceeds accordingly, even though this assumption is violated. Also from control theory are the ideas of *open-loop* and *closed-loop* controllers. The open loop controllers decide the entire sequence of actions to execute before even taking the first action and receive no feedback from the environment as it executes the sequence. When the world is not deterministic, these controller are effectively assuming the system is completely unobservable. The techniques here are closed-loop, since there is a constant feedback signal in the observations received.

6.1 Most Likely State (MLS)

By itself, using a POMDP model and tracking the information state yields a significant amount of information about the system. The information state is the best state estimate we could hope to find and for the task of behaving optimally, it is a *sufficient statistic* [120] for the entire past history of the process. The state with the most probability mass in the information state at a given state, truly is the state that the system is most likely to be in. Assuming we track the information state, the simplest heuristic is to act as if we were in that most likely state. If two or more states are equally likely, we could simply choose one arbitrarily.

With this simple idea, all that is left is deciding which action to execute in state s, when $\forall s' \neq s, b(s) > b(s')$. Since adding partially observability makes the problem hard, we can ignore the partially observability and determine what would be the best action to take for state s if the system was completely observable. Thus, this heuristic makes two assumptions: the system is in the most likely state and that future actions will be based upon the underlying system state.

Recall from Section 2.3 that a POMDP is nothing more than an MDP that lacks direct state information. Therefore, removing the partially observability is simply treating the problem as a COMDP by ignoring the $\mathcal{Z}$ and $\mathcal{O}$ portions of the model. Since solving COMDPs is relatively easy, this presents no real obstacle and the methods of Sections 2.2.3 or 2.2.4 can be applied. Let $\pi_{CO} : \mathcal{S} \rightarrow \mathcal{A}$ be the optimal infinite-horizon COMDP policy for a POMDP. We define the control heuristic policy *most likely state* (MLS) as

$$\pi_{\text{MLS}}(b) = \pi_{CO}(\operatorname*{argmax}_s b(s)) \ .$$

Note that the COMDP policy not only assumes that we know our current state, but that we will also know all of the future current states.

6.2 Action Voting

A potential problem with the MLS control strategy is its complete neglect of all but a single state. Consider a simple three-state, two-action POMDP where the optimal COMDP policy is given by

$$\pi_{CO}(s_0) = a_0 \tag{6.1}$$

$$\pi_{CO}(s_1) = a_0 \tag{6.2}$$

$$\pi_{CO}(s_2) = a_1 \tag{6.3}$$

and suppose the current information state is $b = [\ 0.3 \ \ 0.3 \ \ 0.4\ ]$. The MLS scheme will choose action a_1 despite the fact that we are more likely to be in a state where action a_0 is the best action.

This motivates the *action voting* (AV) control strategy, which assigns a probability distribution over the actions instead of over the states. Again, we solve the POMDP as if it were a COMDP and define

$$w_a(b) = \sum_s b(s) I(\pi_{CO}(s), a) \quad , \tag{6.4}$$

where I is an indicator function as defined in Equation 2.15. Then the AV control strategy becomes

$$\pi_{\mathrm{AV}}(b) = \operatorname*{argmax}_a w_a(b) \quad .$$

This basic voting idea was first used by Simmons and Koenig [113], but they used a planning algorithm based upon a model with deterministic transitions to compute the best action for each state instead of solving the underlying COMDP. For many of the robot navigation domains to be discussed later, these yield essentially the same policies, but in general they can be very different.

6.3 Q-MDP

The AV control strategy is not always the best solution either. The deficiency in the AV scheme lies in it insensitivity to the differences in the actions' values. Recall from Section 2.2.1 that each policy has an associated value function, $V_\pi(\cdot)$ defined over the set of states. Additionally, for each action, there is a related function, $V_\pi^a(\cdot)$, which defines the value of immediately taking action a and following the policy π thereafter.

Consider again a simple three-state, two-action POMDP whose optimal policy has

$$\begin{array}{ll} V_{CO}^{*,a_0}(s_0) = 5 & V_{CO}^{*,a_1}(s_0) = 4 \\ V_{CO}^{*,a_0}(s_1) = 5 & V_{CO}^{*,a_1}(s_1) = 4 \\ V_{CO}^{*,a_0}(s_2) = 0 & V_{CO}^{*,a_1}(s_2) = 10 \end{array}$$

as its value functions. This would yield the policy

$$\pi_{CO}(s_0) = a_0$$

$$\pi_{CO}(s_1) = a_0$$

$$\pi_{CO}(s_2) = a_1$$

and with the information state $b = [\ 0.3 \quad 0.3 \quad 0.4\ ]$ the AV method would select action a_0 since it has a probability of 0.6 of being the best action. However, notice that for the states s_0 and s_1, the alternative action a_1 does not have a much worse value than action a_0. In contrast, there is a significant difference between the two actions in state s_2. In terms of a expectation with respect to the information state, action a_0 will yield a value of $(0.3)(5) + (0.3)(5) + (0.4)(0) = 3$, whereas action a_1 has a value of $(0.3)(4) + (0.3)(4) + (0.4)(10) = 6.4$. Thus, action a_1 has an expected value that is more than twice that of action a_1.

This example leads directly to the Q-MDP control heuristic. This name is derived from the fact that the single action value functions $V^a(s)$ have historically been called *Q-functions*. The Q-MDP control strategy begins, similarly to the MLS and AV strategies, by solving the underlying COMDP. However, in the Q-MDP method we are interested in the Q-functions of the optimal policy rather than the policy itself. The Q-MDP control strategy is given by

$$\pi_{\mathrm{Q-MDP}}(b) = \operatorname*{argmax}_a \left(\sum_s b(s) V^{*,a}(s) \right) \quad .$$

One interesting aspect of this control strategy, is that if the system were to become completely observable after the current action choice (i.e., uncertainty existed for only a single step), the Q-MDP method would yield the optimal strategy.

However, this is also a problem with the Q-MDP method, since it assumes that whatever uncertainty exists will disappear after executing one action. Thus, if an action is available which is fairly neutral in terms of rewards, it has the tendency to choose this action, since it is expecting to be able to do quite well after this step. If this action also does not do much to disambiguate the state, then it leaves the system in the same qualitative state as before (confused, but expecting to be unconfused after the next action) and the Q-MDP method will choose this same neutral action. Assuming this action does little to change the state or reap rewards, the Q-MDP strategy fall victim to its fallacious assumption.

6.4 Dual Mode Control

A problem with the previous strategies is their application in situations where there is a lot of uncertainty in the information state. At this point the Q-MDP, as well as the MLS and AV strategies can begin to make arbitrary choices, especially if the differences between the probabilities or values are minimal. In addition, the assumptions they have about complete observability steer them away from actions which have no reward value, but that may give informational value by reducing the uncertainty in the information state. These two problems can combine to cause these COMDP-based schemes to degrade into a constant cycle of random action selection, never receiving much reward and never doing much to disambiguate the current state.

There is a general concept, known as *dual control*, from the research on adaptive control [4, 60] that concerns itself with the tradeoff between the *control objective* and the *parameter estimation objective.* With adaptive control, on the one hand, there is the objective function which we would like to optimize, but since there is uncertainty in the state or the model, there is the sometimes conflicting objective of trying to estimate the state. In reinforcement learning there is the same problem where it is commonly referred to as the *exploitation vs. exploration* problem [53]. Systems that explicitly attempt to trade off these two objectives would generally be considered dual controllers.

For the case of a POMDP controller, we know the model, so there is not a problem learning the model parameters and the quantity we want to estimate is the current state. Although the information state would seem to solve the state estimation problem, it only does so in a limited way. It gives us a probability distribution over the set of states, but that is not always adequate. The MLS scheme solves the state estimation problem by always selecting the most likely state, but it is possible that there are many states with roughly the same probability. If these states require different actions, then as long as the probabilities of these states remain roughly the same, the system will perform essentially randomly. THe MLS heuristic has no explicit way to steer the information state into situations where one action choice has a higher chance of being better than another.

In this section we describe a class of techniques that could be classified as dual control schemes, since they have two control objectives. The first objective is to take actions that will yield the highest rewards. The second objective is to reduce the *entropy* of the information state. The entropy is a measure of a probability distribution that reflects how spiked or spread out the probability mass is, essentially capturing the amount of uncertainty with a single number. If $f(\cdot)$ is a discrete probability mass function, the entropy of a defined as

$$H(f) = -\sum_{x} \log(f(x))f(x) \ . \tag{6.5}$$

For the discrete information-state case, the entropy is minimized at zero when all the probability mass is on a single state; i.e., there is complete certainty about the current state. The entropy is maximized for the uniform distribution; i.e., $\forall s, b(s) = 1/|\mathcal{S}|$.

The idea behind trying to explicitly reduce the entropy in the information state is to drive the system into a state where the action choice has a higher probability of being the correct choice. In an *entropy reduction* scheme of this sort, we will want to know which action will result in the information state with the lowest entropy. To do this we define the *expected state entropy* of an action and information state to be

$$\mathrm{SH}(b, a) = \sum_{z} \sigma(b, a, z) H(b_z^a) \ . \tag{6.6}$$

Recall from our motivation for the AV control strategy that simply looking at the distribution on the states does not always tell the whole story. In selecting a control action, it can be more important to look at the function $w_a(b)$ in deciding on a proper action. Thus we can define the *action entropy* of an information state as $H(w_a(b))$ and likewise the *expected action entropy* as

$$\mathrm{AH}(b,a) = \sum_z \sigma(b,a,z) H(w_a(b_z^a)) \quad . \tag{6.7}$$

Both Equations 6.6 and 6.7 define a very myopic view of reducing the entropy; It only considers the next step. We could define an n-step entropy reduction scheme which looks multiple steps into the future and considers the longer term entropy. However, since the branching factor of this look-ahead is the number of observations, this method quickly becomes computationally expensive, though clever heuristic pruning or sampling techniques could be used. We will only consider a single step look-ahead for our definitions of expected entropy.

The simplest scheme involving the entropy reduction concept is to define an entropy threshold, κ and have two controlling strategies; one for when the entropy is on each side of the threshold. When the entropy is below this threshold, we can employ a control scheme that tries to maximize the rewards received; e.g., MLS or Q-MDP. When the entropy is above the threshold, we can use either

$$\pi(b) = \operatorname*{argmin}_a \mathrm{SH}(b,a) \quad ,$$

or is equivalent expected action entropy counterpart, to find the action with the lowest expected resulting entropy.

It will be convenient to define the *normalized entropy* of a discrete probability mass function, $f(x)$ as

$$\overline{H}(f) = \frac{H(f)}{H(u)} \ , \tag{6.8}$$

where u is a function representing the uniform distribution over the domain of f; i.e., $\forall x, u(x) = 1/|\mathcal{S}|$. Since the highest entropy is achieved for the uniform distribution, the normalized entropy will always have the range $0 \leq \overline{H}(f) \leq 1$.

Formally, we now define the dual mode control (DM) as

$$\pi_{\text{DM}-\text{X}}(b) = \begin{cases} \text{argmin}_a \, \text{SH}(b,a) & \text{if } \overline{H}(b) > \kappa \\ \pi_{\text{X}}(b) & \text{otherwise} \end{cases}$$

and its related action entropy counterpart as

$$\pi_{\text{ADM}-\text{X}}(b) = \begin{cases} \text{argmin}_a \, \text{AH}(b,a) & \text{if } \overline{H}(w_a(b)) > \kappa \\ \pi_{\text{X}}(b) & \text{otherwise.} \end{cases}$$

The X subscript can be replaced by any other heuristic yielding an entire class of heuristics which will be referred to as DM-X and ADM-X; e.g., DM-Q-MDP, ADM-MLS.

6.5 Weighted Entropy Control

The main problem with the dual-control entropy-reduction schemes of the previous section lies in the complete insensitivity to rewards when the entropy is above the threshold κ. It could be that the action that leads to the lowest expected entropy is considerably worse, in terms of reward, than any of the other actions we might choose based upon a high entropy information state. For a somewhat extreme example, if the very costly action of self-destructing is available and highly reliable, then when the entropy becomes high, this action would be taken with the dual mode control, since there may be little or no entropy in the outcome; e.g., with probability 1 the system is in the state of being destroyed.

What is missing in the entropy reduction scheme is something that relates the rewards of the model to the entropy of the information state. There is no concept of how much the entropy in an information state is worth in terms of the rewards of the model. The heuristic described in this section is an attempt to relate the entropy to the rewards to give some rough measure of the *value of information.* It also attempts to overcome the problem with the Q-MDP method; assuming the uncertainty will go away after one step.

Considering the information state probabilities, when the normalized entropy is zero, then there is no uncertainty in the state. If this certainty persists for the remaining steps, then optimal behavior can be achieved by using the actions specified for the underlying COMDP. At the other extreme, when the normalized entropy is near 1, we have complete uncertainty about the state. If this situation persists, then the future observations are not helping to reduce the uncertainty. When there are no observations, or the observations give no hint about the underlying states, we have a completely unobservable MDP (CUMDP). If we solve an MDP assuming complete unobservability, using V^a_{CU} to denote the related Q-functions, we arrive at a lower bound on the values for a POMDP. Thus, the COMDP and CUMDP solutions provide upper and lower bounds on a POMDPś optimal value function.

Motivated by these arguments, the expression

$$\overline{H}(b)(b \cdot V^a_{CU}) + (1 - \overline{H}(b))(b \cdot V^a_{CO})$$

gives a value for performing action a in an information state based upon the normalized entropy and the upper and lower bounds on the POMDP value function. Unfortunately, there are some problems with using this expression. The most critical problem is that computing V_{CU} exactly is hard[1]. However, there are many ways in which lower bounds can be computed [75, 46] which can be used in place of V_{CU}. We define V_L to be any value function which is a lower bound for the POMDP value function.

Another problem to address is the relative quality of the upper and lower bounds. If one is a fairly loose bound and the other somewhat better, then the simple normalization of the entropy would lead to values skewed towards the loose bound. To compensate for this and allow some bias about the relative quality of the bounds, we introduce the parameter k and define

$$\tilde{H}(f) = \overline{H}(f)^k = \left(\frac{H(f)}{H(u)}\right)^k \quad , \tag{6.9}$$

as a normalized and scaled entropy. This leads to establishing the value of an action for an information state, which we define as the *weighted entropy* (WE) heuristic, with

$$V^a_{\mathrm{WE}}(b) = \tilde{H}(b)(b \cdot V^a_L) + (1 - \tilde{H}(b))(b \cdot V^a_{CO}) \quad .$$

The control strategy would be $\pi_{\mathrm{WE}}(b) = \mathrm{argmax}_a V^a_{\mathrm{WE}}(b)$. The natural extension of this heuristic for the case where we consider the action entropy is

$$V^a_{\mathrm{AWE}}(b) = \tilde{H}(w_a(b))(b \cdot V^a_L) + (1 - \tilde{H}(w_a(b)))(b \cdot V^a_{CO}) \quad .$$

[1] Computing the optimal solution to a CUMDP is NP-complete [95, 90] as are computing weak approximations to CUMDP [20].

6.6 Approximate Value Iteration

An intriguing heuristic uses the exact algorithms discussed in Chapter 3 and value iteration to get approximate solutions which can be used to control a POMDP. Although this can be approached in a much more disciplined way, in this thesis we use an ad-hoc approximate value iteration scheme (APPROX-VI) to generate a set of vectors and evaluate the resulting set of vectors as an approximate version of the value function.

There are two approaches to doing approximate value iteration on an infinite-horizon MDP problem:

- Do each DP stage exactly, but stop after some finite number of stages, i.e., use a finite-horizon solution as the approximation to the infinite-horizon problem;
- Do each DP stage approximately and stop when some comparison criteria on successive stages is met.

For the first case, actual bounds can be placed on the quality of the solution for both finite [9] and continuous space COMDPs [108], making them applicable to the POMDP problems. For the second case, one can also put bounds on the approximation, where the error and the discount factor provide a limit on how wrong the values can be [26, 102, 140].

We do not undertake a disciplined approach to this problem, but our implementation makes doing some form of an approximate DP stage readily available. In order to better quantify the effects of our approximations, we would need to more closely analyze both the algorithms and our implementation in terms of where the approximations are being made and how the errors can be propagated. Nevertheless, we will use the undisciplined APPROX-VI approach which does both approximate DP stages and truncates the number of stages.

The approximation level has an impact on the size of the resulting sets, which in-turn has an affect on the running times of the algorithms. The ad-hoc approach stems from trying various approximation levels until the value function representation sizes were manageable enough to allow value iteration to proceed a significant number of stages. The APPROX-VI scheme is mainly provided as simply another lower bound on the optimal solution, though it also hints at the effectiveness of the exact algorithms when adjusted to allow approximations. Doing this is a more disciplined manner is an interesting area of future research.

6.7 Heuristics Empirical Results

In this section we present three sets of empirical comparisons of the heuristics discussed in the previous sections. We begin by evaluating the performance of these heuristics on some extremely small toy problems. The purpose of this comparison is to establish some relationship between the heuristics and the optimal answers. The difficulty of computing optimal answers requires these problems to be small.

Our next empirical evaluation is driven by a real application and addresses the usefulness of the POMDP model and heuristics for moderately sized problems. The task is one of making navigation decisions in a fully autonomous robot and the POMDP model and these heuristics were used to control an actual robot. Our main focus here is to show that some of these simple heuristics perform quite reasonably in these particular kinds of domains, which demonstrates the usefulness of the POMDP model despite the negative results from the computational complexity vantage.

The conclusions about the heuristics we establish in the robot navigation domain are only applicable to that class of problems. Based on these results, we can say nothing about how the heuristics might perform in other settings. The structure of this class of problems may make them more amenable to heuristic solution than other domains. For instance, the robot navigation domains exhibit some nice locality structure in the transitions, and this may aid the heuristics or bias the results toward a particular heuristic.

Our final empirical evaluation of the heuristics is on a few moderately sized synthetic domains constructed for the purpose of these experiments.

However, we have based these models on realistic problems and have tried to sample from a wide array of domains to avoid biasing the results in a specific direction. As with any empirical comparison where there is no access to the optimal answer, care must be exercised in choosing the problem set.

Additionally, in the absence of the optimal solution, to establish some measure of performance, we have included the results from an *omniscient* (OMNI) controller which is able to see the actual state of the process at all times. For the problems where optimal solutions are not known, this provides an upper bound on the quality of the solution. Note however that the OMNI controller can be much better than even an optimal controller, since the optimal controller is still limited by the partial observability of the domain. When heuristics perform much worst than the omniscient controller it is difficult to know how sub-optimal they are, but when they come close to the omniscient controller, we know that they must also be close to optimal.

Nevertheless, although we can derive some conclusions from these results, until a richer set of POMDP models are available and a more comprehensive evaluation is completed, caution must be exercised concerning how these claims might be extended to POMDP problems in general.

We follow with a section containing some discussion and empirical comparisons for those heuristics where there were tunable parameters; DM, ADM, WE and AWE. We briefly present some results and discussion about how changes in these parameters affect the quality of the control. In the comparisons of these parameterizable heuristics against the other heuristics that precedes this parameter exploration, the results we show for these heuristics reflect the best performance over all parameter setting tried.

We then present a brief section comparing these heuristic solutions to the RL/NDP results of the previous chapter and then proved some discussion of related work.

6.7.1 Experimental Set-up

A simulator for the environments was used to generate state transitions, observations and immediate rewards. Each POMDP model had a problem-specific initial information state and the starting state was chosen to be consistent with this distribution. For all, except the robot-navigation experiments, a single trial consisted of a truncated trajectory of 100 simulated steps starting from the initial state. The immediate rewards, appropriately discounted, were added to yield a sample of the total reward. This was repeated for $10,000$ independent trials and the results reported are the averages over all trials. For the robot navigation experiments discussed in Section 6.7.3 the trajectory length was 300 steps and the results are the averages of 250 trajectories. The discount factor is 0.99 for all problems except for the suite of small problems discussed in Section 6.7.2 where it is 0.95.

The execution time for evaluating the heuristics varied from problem to problem based upon the size of the problem, the relative efficiency of the code for simulating the domain and the particular heuristic used. Since there is no training period or any extensive calculations required for any of the heuristics, the execution time is simply the time necessary to simulate $10,000$ trajectories, where the belief state is updated and the heuristic selects an action for each step. For problems without absorbing states, this amounts to $1,000,000$ steps in the simulator and for problems with absorbing states, this will typically be less. Note that unlike the RL/NDP training phase, there is no need to ensure that the absorbing state problems execute the same number of total steps as other problems since a single evaluation instance is a full trajectory.

6.7.2 Small Problems

To provide some initial basis for the validity of the heuristics and to present some cases where we can actually compare the heuristics against the optimal solutions, we ran the evaluation on the small problems that were the basis for some of the comparisons of the exact algorithms in Section 4.9.2 and the RL/NDP algorithms in Section 5.3.2.

Table 6.1 shows the performance of the various heuristics on these small problems and compares them to the performance of the optimal controller, where available. The boxed entries are used to display the results from a two-sample T-test with $p = 0.995$, where the non-boxed entries are significantly worse than the best value. Notice that for some of these problems, the omniscient controller can be far superior to the optimal controller. It is also possible for the optimal controller to yield a lower value than some of the heuristics, since the optimal controller was used in simulation. Additionally, because the trajectories are truncated, the true infinite-horizon optimal values are higher than what the optimal controller gives for its evaluation values.

Note that in Section 6.7.5 we will explore the WE, DM-MLS, DM-Q-MDP, ADM-MLS and ADM-Q-MDP heuristics over a range of parameter settings, but

Heur.	4x3	4x4	CHEESE	PAINT	SHUTTLE	TIGER	NETWORK	NONLIN	SACI
MLS	1.736	$\boxed{3.702}$	$\boxed{3.380}$	-1.754	$\boxed{32.618}$	-892.485	108.527	6.299	-31.762
Q-MDP	$\boxed{1.867}$	$\boxed{3.708}$	$\boxed{3.461}$	2.277	$\boxed{32.755}$	$\boxed{19.753}$	$\boxed{285.097}$	6.295	7.567
AV	1.747	$\boxed{3.659}$	$\boxed{3.464}$	-9.603	$\boxed{32.687}$	-894.403	131.776	6.278	-44.634
WE	$\boxed{1.819}$	$\boxed{3.720}$	$\boxed{3.349}$	0.435	$\boxed{32.715}$	$\boxed{19.641}$	210.211	6.306	-4.139
AWE	$\boxed{1.857}$	$\boxed{3.713}$	$\boxed{3.358}$	-0.093	$\boxed{32.787}$	$\boxed{19.531}$	252.733	6.292	8.500
DM-MLS	1.772	$\boxed{3.717}$	$\boxed{3.464}$	3.106	—	$\boxed{19.843}$	42.294	6.678	-48.345
ADM-MLS	1.760	—	$\boxed{3.443}$	2.017	—	$\boxed{19.745}$	42.827	6.694	—
DM-QMDP	$\boxed{1.812}$	$\boxed{3.708}$	$\boxed{3.466}$	$\boxed{3.127}$	—	$\boxed{19.690}$	187.911	6.688	-56.146
ADM-QMDP	—	—	$\boxed{3.467}$	2.023	—	$\boxed{19.636}$	187.746	6.686	—
APPROXVI	$\boxed{1.883}$	$\boxed{3.702}$	$\boxed{3.468}$	$\boxed{3.250}$	$\boxed{32.678}$	$\boxed{18.422}$	$\boxed{290.624}$	$\boxed{7.158}$	$\boxed{14.817}$
OPTIMAL	N/A	3.712	3.464	3.279	32.700	19.181	290.998	7.158	N/A
OMNI	2.466	4.654	3.910	12.678	32.650	198.816	490.997	12.577	16.904

Table 6.1: The heuristic algorithms on the suite of small problems. T-test with $p = 0.995$.

Table 6.1 shows the best answer among all the parameter settings for those algorithms. For the dual-mode controllers, the entries marked with a "—" indicate that the information state's entropy never exceeded any of the thresholds we tried (the smallest being 0.1). This means that no actions were taken in an attempt to reduce the entropy; e.g., the DM-MLS heuristic performed exactly the same as the plain MLS scheme for the SHUTTLE problem. Section 6.7.5 will discuss these "—" entries in more detail.

An interesting result from Table 6.1 is that the APPROX-VI scheme is never worse than any of the other heuristics and significantly better that all on the NONLIN and SACI domains. This shows the potential for using the exact algorithms of Chapter 3 in approximation schemes, but the drawback here is that the APPROX-VI controller is the result of significantly more computation that the other heuristics. Some amount of time is needed to run the approximate value iteration to produce the answer. The heuristics need to solve the underlying COMDP first, but this time requirement is extremely small compared to executing value iteration approximately. This table also shows that there are domains where all of these heuristics are useful control strategies, sometimes approaching optimal behavior.

There is an interesting reason why the Q-MDP method does better than the MLS or AV schemes on the PAINT and TIGER problems. Recall, that the Q-MDP method is the optimal control strategy if the uncertainty about the state would be removed after the next action. Although the PAINT and TIGER problems do not have this property, it is the case that good results can be obtained on these problems if the controller is willing to take a low cost informative action for a single step. The MLS and AV schemes do not favor these informative action because of their low cost, but the Q-MDP method makes a slightly more informed decision, based on a single step of uncertainty. It turns out that the action chosen, while not completely informative, is useful enough that after one step Q-MDP's subsequent decisions are fairly good ones. If the domain requires a sequence of informative actions, we would expect the Q-MDP method to do poorly. We will see an instance of this in the robot navigation domains to follow.

For the PAINT problem and aside from the APPROX-VI controller, only the two dual-mode controllers approach the optimal performance. The presence of low cost information gathering actions, combined with significant penalties for wrong action choices force these heuristics to continually apply the informative actions until the state is known with sufficient confidence. This type of behavior turns out to be the general structure present in the optimal controller. The entropy-based heuristics do well on the TIGER problem because it has similar structure to the PAINT problem.

Ignoring the APPROX-VI for all of these small problems, there is some heuristic which yields fairly good control policies. However, there is no single heuristic which is universally good, meaning that some exploration would be required to establish which heuristics would be applicable to a particular problem. These small problems do provide a useful forum for trying to characterize the structures of problems for which the various heuristics might do well. However, it could be that the structure that is most important for applying these heuristics is that the problem be small. The following subsections show this not to be the case by applying the heuristics to much larger problems.

6.7.3 Robot Navigation

With no theoretical guarantees on the quality of the heuristic solutions, the usefulness of the heuristics are best determined with their applicability to actual problems. One question that has remained largely unanswered is the feasibility of using the POMDP model in a real planning problem. Most of the previous experimental results have been on either random, toy or synthetic problems. One step we have taken in the direction of applicability is to use a POMDP model on an autonomous robot navigating in an office environment. Although the structure of the office environment has many nice features for applying the the somewhat abstract POMDP model, there is still a high degree of uncertainty in the crude observations available and the action outcomes. The need for algorithms to handle noisy environments is quite noticeable in autonomous robot research [18]. It is the robot navigation domain which motivated the development of these heuristics.

Previous work using POMDP models for planning in mobile robots by Simmons and Keonig [113] on the robot XAVIER used a single heuristic (essentially the voting heuristic) and our research has explored the question of whether there are better heuristics as well as whether POMDP models can be applied to realistic problems. Additionally, the work by Simmons and Koenig used a more intriguing, though more complex, POMDP model of the environment than a simply discretization of the floor area. It is an open question whether the results here are applicable to models constructed with their methods, though we suspect they are.

Another related effort is that by Nourbakhsh *et al.* [94] on the robot

DERVISH. Although not explicitly grounded in an MDP model, they use a probabilistic model to help predict potential resulting states. They also use the certainty equivalence principle and assume the robot is in the most likely state, though their planning schemes tend to be different from that used for the MLS heuristic. More discussion of the similarities and differences between their approach and those presented here can be found in Cassandra, Kaelbling and Kurien [21] along with some empirical comparisons.

One of the shortcomings of all of these approaches in the need for a full explicit model of the environment. A more natural situation is for the robot to uncover the structure and model parameters. We present no solutions to this problem but refer to the work by Koenig and Simmons [59] and Shatkay and Kaelbling [112].

Synthetic Environments

The general robot navigation domain is discussed in Appendix H.5. Here we explore 24 specific POMDP model instances of this domain which correspond to 4 different location configurations, 3 different starting/goal state configurations and 2 different noise models. One difference of these navigation domains from the discussion in the appendix is that being in a non-goal state and choosing the action that declares the goal state results in a transitioning to a zero-cost absorbing state with no immediate reward. For the domains described in Appendix H.5, the outcome is a self-transition and a penalty in the form of negative reward.

We first present the results for a noise model where there is relatively little noise (standard) in the state transitions and observations, which roughly

Action	Standard probabilities
move-forward	N (0.11), F (0.88), F-F (0.01)
turn-left	N (0.05), L (0.9), L-L (0.05)
turn-right	N (0.05), R (0.9), R-R (0.05)
no-op	N (1.0)
declare-goal	N (1.0)
Action	**Noisy probabilities**
move-forward	N (0.05), F (0.7), F-F (0.05), L (0.1), R (0.1)
turn-left	N (0.1), L (0.7), L-L (0.1), F-L (0.1)
turn-right	N (0.1), R (0.7), R-R (0.1), F-R (0.1)
no-op	N (1.0)
declare-goal	N (1.0)

Table 6.2: Action probability specifications for synthetic robot navigation domains.

correspond to those of a real robot. We then explore the effects of adding noise (noisy) to these same 4 configurations. The transition and observation probabilities for the both the standard and noisy noise models are given in Tables 6.2 and 6.3, where Appendix H.5 gives the interpretation of these tables.

The four specific location configurations and three different starting/goal state combinations were selected to provide a range of challenges to the heuristics and to eliminate any odd effects that could arise with a single domain or single configuration of start/goal states. In particular, one set (experiment 1) of starting states reflect the case where the robot knows with certainty its starting location. The next set (experiment 2) is a situation where there is minor uncertainty about the initial state; e.g., the robot could

z_a **Actual**	z_o **Observed**	$P(z_o \mid z_a)$ **Standard**	**Noisy**
wall	wall	0.90	0.70
wall	open	0.04	0.19
wall	doorway	0.04	0.09
wall	undetermined	0.02	0.02
open	wall	0.02	0.19
open	open	0.90	0.70
open	doorway	0.06	0.09
open	undetermined	0.02	0.02
doorway	wall	0.15	0.15
doorway	open	0.15	0.15
doorway	doorway	0.69	0.69
doorway	undetermined	0.01	0.01
undetermined	undetermined	1.00	1.00

Table 6.3: Conditional observation probabilities for synthetic robot navigation domains.

start in one of two possible states. The final set (experiment 3) represents the case where there is complete uncertainty about its initial state; i.e., the initial information state is a uniform distribution over all states in the model.

Figures 6.1 through 6.4 show both the location layout and the various initial and goal states for the three sets of experiments. The dark squares represent locations that are rooms and the lighter shaded squares are corridor or hallway locations. These configurations are loosely based upon actual office buildings and were not constructed to favor any particular control strategy. They all have between 200 and 300 states, 64 observations and 5 actions.

When this evaluation was done, the DM, ADM and WE heuristics were still in an immature state. We have omitted the results of these heuristics and focus on the MLS, Q-MDP and AV heuristics. The DM, ADM and WE heuris-

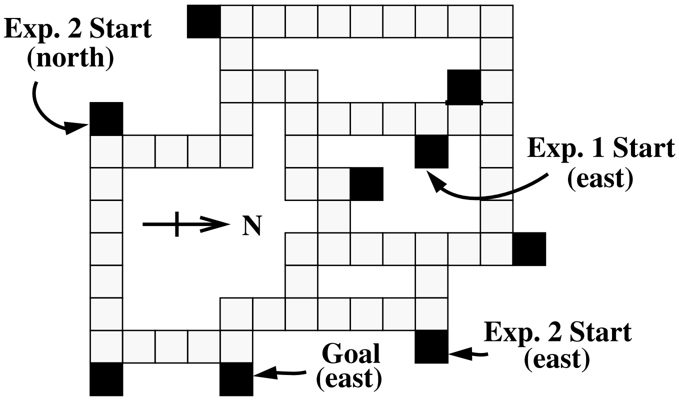


Figure 6.1: Synthetic office environment A.

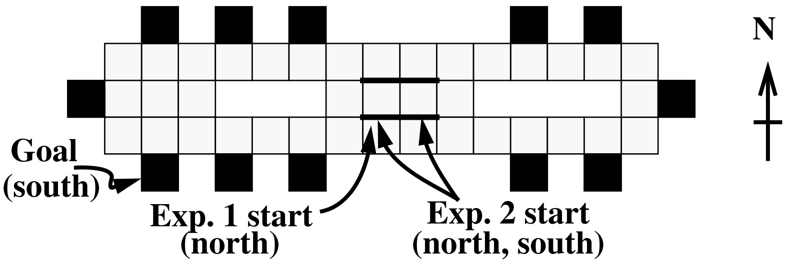


Figure 6.2: Synthetic office environment B.

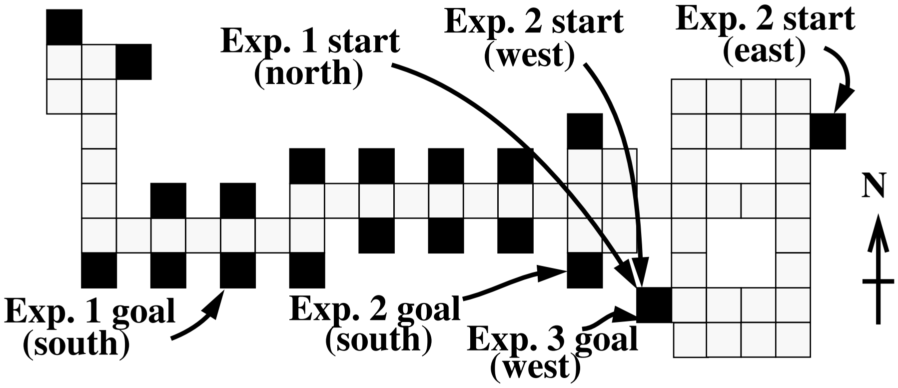


Figure 6.3: Synthetic office environment C.

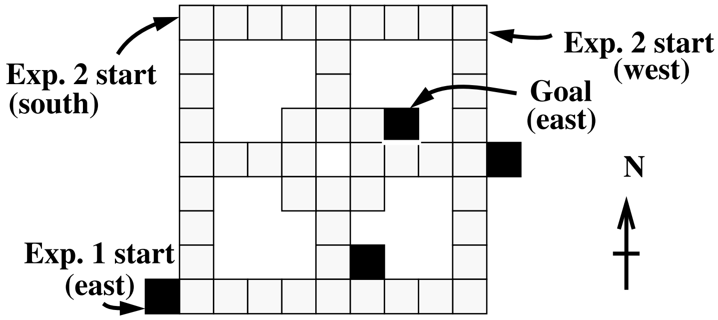


Figure 6.4: Synthetic office environment D.

tics are evaluated in Sections 6.7.4 and 6.7.5 where we also consider some navigation environments similar to the ones in this section. Additionally, there are also some related heuristics which were tried, but never proved superior in any of our experiments. We have eliminated these results as well, and refer the reader to the original paper which introduced the results given here [21].

For the experiments of this section we performed a T-test to determine whether or not there were significant differences between the heuristics. In the tables, the best performance is outlined darkly and those that are not significantly worse are lightly outlined.

Standard Noise Model

Table 6.4 shows the results of the different heuristics on the four synthetic office environments when the starting state is known.

Although there is some variability between the methods, for the most part none of the heuristics do poorly. Since each is an approximation method, there are particular circumstances where they can be made to to perform arbitrarily poorly. The data points where one of the methods ap-

Heuristic	A	B	C	D
MLS	0.642	$\boxed{0.749}$	$\boxed{0.662}$	0.791
AV	0.639	0.704	0.612	0.800
Q-MDP	$\boxed{0.662}$	0.743	0.452	$\boxed{0.825}$
OMNI	0.677	0.846	0.756	0.836

Table 6.4: Experiment 1: Known starting state, standard noise model.

Heuristic	A	B	C	D
MLS	0.695	$\boxed{0.639}$	$\boxed{0.779}$	0.791
AV	0.669	0.000	0.737	0.791
Q-MDP	$\boxed{0.704}$	0.000	$\boxed{0.788}$	$\boxed{0.853}$
OMNI	0.728	0.848	0.845	0.878

Table 6.5: Experiment 2: Multiple possible start states, standard noise model.

pear significantly worse than the others are examples of such circumstances. Note that the MLS method is fairly robust across the environments, showing that not much more than the information captured in the information state is needed to do exceedingly well in this situation.

The next situation we address is when the agent is not certain of its starting state. In the experiments shown in Table 6.5, there are two possible starting states (four possible states for office B) that are similar in their immediate surroundings.

This is the first case where we see some of the methods performing poorly. The Q-MDP and AV methods are never able to reach the goal in the office B experiment. This is because they cycle through the same set of actions without making any progress toward the goal. This cycling behavior is not always present, as can be seen by their performance in the other environments. In fact, the Q-MDP method is significantly better than the

Heuristic	A	B	C	D
MLS	$\boxed{0.630}$	$\boxed{0.615}$	$\boxed{0.601}$	$\boxed{0.729}$
AV	$\boxed{0.599}$	0.570	0.257	0.648
Q-MDP	0.405	0.502	0.308	0.574
OMNI	0.750	0.868	0.853	0.898

Table 6.6: Experiment 3: Uniform starting belief, standard noise model.

others in the office A and D experiments, where that particular configuration of starting states and goal states allows it to behave nearly as well as the omniscient method.

A problem with selecting the AV or Q-MDP methods is that it is not easy to know beforehand if the particular environment will bring out the best or the worst in these heuristics. The MLS method performs reasonably well across all of these situations and would be the preferred choice unless something more was known about the particular problem instance.

The final situation we explore is the most difficult from the control perspective: what if the robot is equally likely to start in any of the states? In this situation, its initial belief distribution is uniform over all states. Table 6.6 shows the results of applying the heuristics to this situation. These results show that the Q-MDP and AV methods do not usually perform well when the uncertainty is high. The MLS method is still quite robust and suggests that for moderately noisy environments, it is the best choice across different types of environment layouts and starting beliefs.

Noisier Noise Model

The navigation problem becomes harder as the noise in action and observation increases. In order to gauge the effects of noise on the various methods,

Heuristic	Office A	Office B	Office C	Office D
MLS	$\boxed{0.082}$	$\boxed{0.190}$	$\boxed{0.070}$	0.264
AV	0.044	0.156	0.025	$\boxed{0.307}$
Q-MDP	0.000	0.000	0.000	0.000
OMNI	0.576	0.782	0.654	0.779

Table 6.7: Experiment 1: Known starting state, noisy noise model.

Heuristic	Office A	Office B	Office C	Office D
MLS	0.166	$\boxed{0.150}$	$\boxed{0.181}$	0.251
AV	$\boxed{0.206}$	0.124	0.146	$\boxed{0.293}$
Q-MDP	0.000	0.000	0.000	0.000
OMNI	0.650	0.792	0.784	0.830

Table 6.8: Experiment 2: Multiple possible start states, noisy noise model.

we repeated the experiments using the increased noise (noisy) action and observation probabilities shown previously in Tables 6.2 and 6.3. Tables 6.7, 6.8 and 6.9 show the results of these experiments.

The Q-MDP method was universally bad across all configurations and starting beliefs. In none of the trials did it ever declare itself to be in the goal. This is a direct result of Q-MDP's assumption that it will be completely disambiguated on the next step. Since the noise is high, it will never have a very confident belief that it is in the goal. It would prefer to delay declaring the goal by doing one more action in hopes of knowing where it will be after

Heuristic	Office A	Office B	Office C	Office D
MLS	0.130	0.125	$\boxed{0.122}$	0.236
AV	$\boxed{0.168}$	$\boxed{0.183}$	0.082	$\boxed{0.297}$
Q-MDP	0.000	0.000	0.000	0.000
OMNI	0.660	0.811	0.774	0.852

Table 6.9: Experiment 3: Uniform starting belief, noisy noise model.

that action. Among the MLS and AV heuristics, neither is clearly superior to the other, though the MLS heuristic seems to be more applicable when the initial uncertainty is low.

Real Robot Navigation Environment

The four synthetic environments of the previous sections lay the basic foundation for the performance of the heuristics in navigation domains. However, the ultimate aim is to use POMDP models on a fully autonomous robot, in a real office environment. We would like the conclusions from the previous evaluations on synthetic environments to be applicable to real robot navigation problems. We therefore took a two-stage approach to connecting those results to actual autonomous robot navigation.

The first stage consists of building a model of the environment where the robot needs to navigate and evaluating the heuristics on this model in simulation. This bridges the gap between the synthetic environments and a real environment, though both are evaluated using simulations. The second stage is to the evaluate the heuristics with the actual robot using this model. This two-pronged approach is necessitated by the fact that it is extremely time consuming to gather the data using the actual robot, rendering hundreds of trials impractical. Moving directly from simulated experience on synthetic environments to real experiences on a real environment would leave open the question of whether or not the results were an artifact of the model or the real experiences. By running hundreds of simulations on the same model, we eliminate one of the variables.

Figure 6.5 shows the real office environment navigated by the robot which

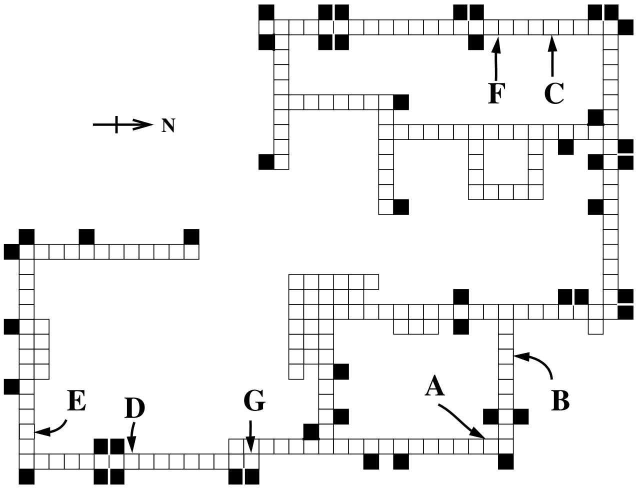


Figure 6.5: Real office environment.

consists of 1,052 POMDP states. Due to the size of this domain, we used 7 different configurations of initial information states. For all of these experiments, the goal state was the same, location G, facing east. The first three experiments were for a known starting location, (A-east, B-east and C-south), roughly corresponding to differing physical distances from the goal. The next three experiments used a starting belief over two locations; for each of the previous starting states we added a state, (D-east, E-east and F-north respectively), roughly the same distance and with similar immediate observations. The final experiment was conducted with a uniform starting belief over all states. We used the same standard noise model that was used in the synthetic domain experiments.

Table 6.10 shows the simulation results for 100 trials. These results are consistent with the synthetic office A layout, which is a simplified version of this real environment.

The last step is to connect the results of Table 6.10 to the actual robot

Alg.	Experiment 1	2	3	4	5	6.	7
MLS	0.774	0.717	0.452	0.776	0.763	0.390	0.504
AV	0.690	0.721	0.339	0.710	0.775	0.207	0.539
Q-MDP	0.797	0.730	0.454	0.817	0.755	0.428	0.407
OMNI	0.809	0.746	0.489	0.855	0.780	0.481	0.652

Table 6.10: Simulations of real robot office environment, standard noise model.

experiments. Although we give the results in Table 6.11, we omit the details of these experiments, the actual experimental set-up and the overall design of the controlling architecture of the robot, though we note that there are many interesting implementation issues which arose from this project. The interested reader is referred to the original paper [21], however there are a few points to be made about Table 6.11[2]:

- the number of trajectories run for each experiment was extremely small: three for experiments 1, 2 and 3, and six for experiments 4, 5 and 6 (three for each of the two starting locations);
- experiment 7 was not run, since properly evaluating for the uniform starting location would require hundreds of trajectories from a sampling of all possible states;
- the data roughly agrees with the simulated results.

Aside from the general usefulness of the POMDP model and these heuristics in the robot navigation domain, this work demonstrated that the flexibility in the high-level POMDP model simplifies the task of the low level

[2]The experiments on the robot are the product of the efforts of James Kurien with the autonomous robot Ramona.

Alg.	Experiment 1	2	3	4	5	6
MLS	0.83	0.77	0.49	0.84	0.78	0.31
AV	0.86	0.77	0.50	0.00	0.77	0.00
Q-MDP	0.83	0.78	0.46	0.78	0.76	0.25

Table 6.11: Experiments on robot.

control design. Typically, designers of low-level controlling software spend a painstaking amount of time assuring reliability across all possible situations which could be encountered. The POMDP model allows for the imperfection of the low-level control, which can even be combined with the true non-determinism of the environment. As long as a reasonable failure model can be built and adhered to by the low-level controller, the high-level model can recover from the errors, whether they be from the true stochasticity of the environment or artifacts of the low-level implementation.

6.7.4 Other Domains

In an effort to extend the small problem and robot navigation conclusions, we have evaluated the heuristics (including the parameterized heuristics WE, AWE, DM and ADM) on a suite of larger synthetic problems. These domains were discussed in Section 5.3.5 and are detailed in Appendix H. Note that the CIT, MIT, SUNYSB, PENTAGON and FOURTH domains are very similar to the five navigation domains presented in the previous section, where there is a single known starting state. Since the entropy-based heuristics were not explored in the previous discussion, this will allow us to gauge their effectiveness in domains with similar structure. We will also explore variations of these navigation domains where the initial information state is uniform across all locations: CIT-U, MIT-U, SUNYSB-U, PENTAGON-U and FOURTH-U

Although the problems in this sub-section range from 100 to 8,000 states, this is not the limit of the applicability of these heuristics. COMDPs that are larger than this can be solved and information state vectors larger than this can be represented. However, as the state and observation spaces increase the time requirements for the information state update could begin to matter. Note that the action space also has some bearing upon this, though we limit our discussion to the cases where the size of state space usually dwarfs the size of the action space.

We first apply the heuristics to the the 57 and 89 state domains used in the Chapter 5, and which were shown in Figures 5.2 and 5.3. Although both domains have the property of complete initial uncertainty, HALLWAY has a number of landmarks and an asymmetrical layout, helping to disambiguate the location, even for a random sequence of actions. On the other hand, HALLWAY-2 has no distinguishing landmarks, and the property of being hideously symmetrical. Table 6.12 shows that again, the APPROX-VI heuristic is significantly better than the rest, establishing again the potential for the use of exact algorithms in approximation schemes. Besides APPROX-VI we see that the WE and AWE heuristics do better in the HALLWAY-2 environment, where the information state entropy is always likely to be high.

Next, we evaluate the full set of heuristics on the robot navigation environments. Table 6.13 shows the results for the 5 domains where the initial state is know and Table 6.14 shows the the same domains but where there is complete uncertainty in the initial information state, i.e., maximum entropy.

Heur.	HALLWAY	HALL.-2
MLS	0.802	0.174
Q-MDP	0.344	0.097
AV	0.756	0.084
WE	0.598	0.326
AWE	0.776	0.378
DM-MLS	0.818	0.213
ADM-MLS	0.823	0.215
DM-QMDP	0.598	0.193
ADM-QMDP	0.602	0.224
APPROXVI	$\boxed{1.001}$	$\boxed{0.416}$
OPTIMAL	N/A	N/A
OMNI	1.519	1.189

Table 6.12: The heuristic algorithms on the 57 and 89 state problems. T-test with $p = 0.995$.

Heur.	CIT	MIT	SUNY.	PENT.	FOURTH
MLS	$\boxed{0.804}$	$\boxed{0.858}$	$\boxed{0.764}$	0.789	$\boxed{0.587}$
Q-MDP	$\boxed{0.832}$	0.812	$\boxed{0.759}$	$\boxed{0.821}$	$\boxed{0.590}$
AV	$\boxed{0.807}$	$\boxed{0.863}$	0.711	$\boxed{0.795}$	$\boxed{0.586}$
WE	$\boxed{0.834}$	0.814	$\boxed{0.760}$	$\boxed{0.822}$	$\boxed{0.592}$
AWE	$\boxed{0.833}$	0.812	$\boxed{0.760}$	$\boxed{0.822}$	$\boxed{0.592}$
DM-MLS	—	—	—	—	—
ADM-MLS	—	—	—	—	—
DM-QMDP	—	0.815	—	—	—
ADM-QMDP	—	—	—	—	—
APPROXVI	0	0	0	0	0
OPTIMAL	N/A	N/A	N/A	N/A	N/A
OMNI	0.845	0.894	0.809	0.836	0.625

Table 6.13: The heuristic algorithms on some robot navigation problems. T-test with $p = 0.995$.

Heur.	CIT-U	MIT-U	SUNY.-U	PENT.-U	FOURTH-U
MLS	0.626	0.524	0.508	0.682	0.451
Q-MDP	0.366	0.556	0.330	0.573	0.340
AV	**0.654**	**0.594**	**0.575**	**0.738**	**0.502**
WE	0.368	0.559	0.334	0.577	0.349
AWE	0.373	0.557	0.336	0.580	0.348
DM-MLS	0.629	0.480	0.495	0.691	0.440
ADM-MLS	—	—	0.492	0.692	—
DM-QMDP	0.370	0.556	0.330	0.580	0.347
ADM-QMDP	—	—	—	0.570	—
APPROXVI	0.000	0.022	0.008	0.019	0.007
OPTIMAL	N/A	N/A	N/A	N/A	N/A
OMNI	0.837	0.893	0.845	0.896	0.652

Table 6.14: The heuristic algorithms on the robot navigation problems with uniform initial information state problems. T-test with $p = 0.995$.

For the case of known initial state, we see first, the dual mode controllers are rarely of any help. The exception is the drastically symmetric MIT domain, and even then its performance is below what is obtainable with other methods. The second result is that all the other heuristics do more or less equivalently. By looking at the omniscient controller's value, we can also see that they must not be too far off from the optimal behavior. As in the previous navigation experiments, we see that very unsophisticated techniques do quite well in domains with this structure. Finally, we see that the APPROX-VI heuristic did miserably in these domains. The sizes of the domains limited the precision and length with which we could obtain results through the approximate value iteration and the final answers appear to be poor controllers for these domains.

In Table 6.14, where the initial state uncertainty is maximized, we get the surprising result that the AV method is significantly better than all the other heuristics, except in the CIT-U domain. This is quite a contrast to the known initial state case and very similar to the previous results shown for the other robot navigation environments where there was an uncertain initial state and noisier actions. Notice that the APPROX-VI controller actually does better in this situation than it did in the known starting state location. This is attributable to the randomness in the initial state, since there are some number of times where the robot will start very close to the goal. In this case, even random behavior is likely to lead one to the goal, though it still is required to choose the `declare-goal` action to get a reward.

The last set of domains we use to explore the usefulness of the heuristics have their results shown in Table 6.15. We see the resurgence of the APPROX-VI controller, though the size of the BASEBALL domain resulted in VI taking considerably longer than we were willing to wait to produce even a single stage at a crude approximation. This is more of a problem with our particularly unmotivated method for doing approximate value iteration, than it is with the idea itself. For the most part, the results on this set of domains are mixed. For a given domain there are a number control heuristics which do well. However, the AV heuristic is the only one which is always among the best.

Of these domains, only the IFF and MACHINE domains have explicit information gathering actions. Because of this, we would expect the MLS, Q-MDP and AV heuristics to not do so well here, since they are based upon assumptions of complete observability. The entropy-based heuristics would seem to have an advantage in these domains, though the results in these domains do not bear this out. Although DM-MLS is the best for the IFF domain,

Heur.	IFF	BB	MACHINE	ALOHA10	ALOHA30
MLS	$\boxed{8.191}$	$\boxed{0.634}$	57.042	$\boxed{126.082}$	$\boxed{852.773}$
Q-MDP	$\boxed{4.496}$	0.101	$\boxed{59.693}$	$\boxed{127.429}$	$\boxed{851.035}$
AV	$\boxed{8.004}$	$\boxed{0.633}$	$\boxed{57.901}$	$\boxed{126.227}$	$\boxed{849.109}$
WE	3.452	$\boxed{0.658}$	37.196	110.702	751.993
AWE	3.464	$\boxed{0.668}$	53.877	110.813	736.230
DM-MLS	$\boxed{8.389}$	—	—	$\boxed{126.296}$	—
ADM-MLS	—	—	—	$\boxed{124.652}$	—
DM-QMDP	—	0.350	—	—	—
ADM-QMDP	5.684	0.461	—	$\boxed{125.947}$	—
APPROXVI	$\boxed{4.590}$	N/A	$\boxed{59.884}$	$\boxed{126.808}$	$\boxed{848.830}$
OPTIMAL	N/A	N/A	N/A	N/A	N/A
OMNI	10.079	0.658	66.236	145.572	937.143

Table 6.15: The heuristic algorithms on the other large problems. T-test with $p = 0.995$.

MLS is not significantly worse and the MACHINE domain has Q-MDP being among the best. However, for those two domains, we see that often, the dual-mode controllers never require taking entropy-reduction actions. This hints that although these problems have information gathering actions, they are seldom required. We speculate that problems with information gathering actions and that tend toward highly entropic information states would be the type of structure where the entropy-based schemes would be significant improvements to the MLS, Q-MDP and AV heuristics. Although one could construct models with just this structure, we have not explored this possibility, since we want to avoid specially constructed domains.

Although the results are as mixed as they were on the suite of small problems, this does show that for each domain there is some heuristic that does fairly well. For some, the quality of the heuristic compared to the OMNI controller shows that there is little room for improvement. For others, there appears to be room for improvement, but this cannot really be assessed without access to the optimal controller, since the OMNI heuristic can be significantly better than even the optimal POMDP control.

6.7.5 Parameterized Heuristics

In the preceding evaluations of the heuristics, the DM, ADM, WE and AWE heuristic results reflected the best results from the range of parameter settings used. In this section we present results for all of the parameter settings used and discuss the implications of adjusting these parameters.

Dual Mode

The DM and ADM heuristics are parameterized by an entropy threshold, κ. We ran the heuristics DM-MLS, ADM-MLS, DM-Q-MDP and ADM-Q-MDP on every domain 9 times and used a different threshold each time, where the range of the thresholds was 0.1 to 0.9 in increments of 0.1. As a typical result, Table 6.16 shows the results for the DM-MLS heuristic.

Domain	**Threshold** 0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
4x3	1.416	1.420	1.489	1.527	1.767	1.772	1.741	1.760	1.761
4x4	3.487	3.487	3.494	3.491	3.489	3.491	3.654	3.653	3.717
CHEESE	3.216	3.216	3.462	3.464	3.440	3.434	3.434	3.439	3.445
PAINT	1.648	2.117	2.783	2.771	3.106	-1.753	-1.771	**-1.763**	**-1.773**
SHUTTLE	**32.632**	**32.654**	**32.743**	**32.606**	**32.724**	**32.625**	**32.708**	**32.756**	**32.602**
TIGER	16.178	19.837	19.132	19.843	19.378	19.291	-72.041	-72.892	-73.513
NETWORK	-595.119	-595.184	-595.213	-595.132	-595.298	-435.664	-414.192	-254.913	42.294
NONLIN	6.659	6.678	6.665	**6.275**	**6.287**	**6.274**	**6.288**	**6.314**	**6.284**
SACI	-81.993	-82.350	-81.881	-82.293	-81.982	-81.655	-68.306	-49.446	-48.345
HALLWAY	0.255	0.487	0.534	0.780	0.786	0.818	0.804	0.805	0.809
HALL.-2	0.017	0.053	0.118	0.182	0.196	0.203	0.213	0.212	0.212
CIT	-58.076	-0.095	0.753	**0.805**	**0.808**	**0.802**	**0.805**	**0.803**	**0.806**
MIT	-0.164	0.799	0.857	**0.857**	**0.857**	**0.859**	**0.855**	**0.859**	**0.860**
SUNY.	-9.648	0.266	0.742	**0.762**	**0.765**	**0.766**	**0.765**	**0.764**	**0.765**
PENT.	-5.179	0.160	0.647	0.754	**0.791**	**0.792**	**0.791**	**0.796**	**0.793**
FOURTH	-45.376	0.007	0.560	**0.588**	**0.586**	**0.588**	**0.586**	**0.587**	**0.585**
IFF	0.370	8.181	8.138	8.389	8.061	8.299	8.193	7.948	8.154
BB	0.343	0.401	**0.649**	**0.627**	**0.618**	**0.640**	**0.605**	**0.623**	**0.634**
MACHINE	-408.104	-286.265	-123.941	-41.377	5.963	21.595	33.916	54.255	**56.923**
ALOHA10	92.855	97.556	105.587	112.436	118.920	126.296	125.766	125.814	125.662
ALOHA30	683.531	688.004	693.332	705.423	759.159	842.663	**849.878**	**848.449**	**848.477**
CIT-U	-26.739	-16.279	-6.524	-0.052	0.050	0.359	-0.080	0.615	0.629
MIT-U	-22.874	-13.485	-6.614	-6.050	0.363	0.375	0.480	0.478	0.466
SUNY.-U	-27.403	-19.702	-15.305	-11.571	-5.309	0.384	0.473	0.495	0.487
PENT.-U	-27.203	-26.343	-13.758	-0.909	-0.320	-0.716	-1.435	0.686	0.691
FOURTH-U	-37.774	-18.168	-11.376	-6.123	-3.852	-0.393	-0.478	0.438	0.440

Table 6.16: Threshold values and the DM-MLS heuristic. T-test with $p = 0.995$.

This large table has entries which are visually marked in one of four ways

- Entries that are white text on a black background indicate that the information state entropy *never* exceeded this threshold. If none of the steps taken were to reduce the entropy, then the heuristic simply degenerates into either the MLS or Q-MDP heuristic. The full results showing the percentage of steps for which the threshold was exceeded are given in Appendix I.2, Tables I.13 through I.16.
- Entries with a thick black box indicate that at least once the entropy exceeded the threshold level *and* it is the best value for that domain across all threshold settings.
- Entries with a thinner black box indicate that this value is not significantly worse than the best value.
- Entries with no markings are either statistically worse than the best entry, or it is the case that the best entry for that domain occurred when no entropy reduction actions were needed.

The most dramatic results concerning the threshold value occur on the PAINT and TIGER problems, where the wrong threshold can result in exceedingly poor performance. Aside from this, the dual mode controllers seem to do best when the threshold is relatively high. In other words, relying on a certainty equivalent type controller is a good heuristic, unless the entropy is very high.

The varying threshold results for the ADM-MLS, DM-Q-MDP and ADM-Q-MDP heuristics can be found the Appendix I.2, Tables I.10 through I.12.

Unfortunately, there is no clear-cut best value of κ, and the actual value seems to be domain dependent. Again, we suspect that, for the most part, these problems do not exhibit the structure where the dual-mode controllers would be clearly preferable to the non-entropy-based controllers.

Weighted Entropy

The WE and AWE heuristics have an exponent for scaling the normalized entropy. We ran all the WE and AWE experiments 3 times using the differing exponents of 1/3, 1 and 3 each time. Table 6.17 shows the results for the WE heuristic and Table 6.18 shows the results for AWE. The dark boxed entries highlight the exponent value which gave the best result, with the lighter boxes indicating no significant difference.

For many of the domains, these particular exponents give roughly equivalent performance. However, there are domains where the actual value does matter (e.g., HALLWAY, HALLWAY-2, IFF, BB) and, unfortunately, there is no clear cut best value. Thus, the proper exponent is either domain dependent or some function of the combination of the domain and the particular upper and lower bounds used in the WE and AWE heuristics.

Note also that the particular lower bound we used was to simply evaluate the value function, once for each action, for a policy which always choose that action. These one-action policies are not only easy to compute, but their value functions are linear, making them easy to represent.

Domain	**Exponent** 1/3	1	3
4X3	0.965	1.296	1.819
4X4	3.709	3.708	3.720
CHEESE	3.262	3.246	3.349
PAINT	0.434	0.435	0.417
SHUTTLE	32.656	32.638	32.715
TIGER	-9.888	8.544	19.641
NETWORK	148.960	197.002	210.211
NONLIN	6.283	6.306	6.274
SACI	-4.481	-4.379	-4.139
HALLWAY	0.598	0.513	0.338
HALL.-2	0.326	0.304	0.180
CIT	0.833	0.834	0.833
MIT	0.814	0.810	0.811
SUNY.	0.760	0.758	0.757
PENT.	0.822	0.822	0.821
FOURTH	0.592	0.591	0.592
IFF	3.328	3.452	-12.067
BB	0.634	0.658	0.210
MACHINE	37.196	37.144	37.176
ALOHA10	109.930	110.702	109.621
ALOHA30	689.554	691.697	751.993
CIT-U	0.361	0.364	0.368
MIT-U	0.559	0.556	0.550
SUNY.-U	0.324	0.334	0.329
PENT.-U	0.573	0.577	0.573
FOURTH-U	0.337	0.347	0.349

Table 6.17: Exponent values and the WE heuristic. T-test with $p = 0.995$.

Domain	Exponent 1/3	1	3
4X3	1.373	1.843	1.857
4X4	3.706	3.713	3.713
CHEESE	3.255	3.254	3.358
PAINT	-0.093	-0.224	-0.237
SHUTTLE	32.725	32.787	32.742
TIGER	-9.904	8.587	19.531
NETWORK	150.696	197.650	252.733
NONLIN	6.286	6.253	6.292
SACI	8.500	8.321	7.495
HALLWAY	0.564	0.776	0.340
HALL.-2	0.049	0.378	0.212
CIT	0.833	0.833	0.833
MIT	0.812	0.811	0.810
SUNY.	0.760	0.759	0.759
PENT.	0.821	0.822	0.821
FOURTH	0.591	0.591	0.592
IFF	3.097	3.464	2.264
BB	0.668	0.655	0.401
MACHINE	40.931	46.670	53.877
ALOHA10	110.266	110.813	109.921
ALOHA30	689.189	693.579	736.230
CIT-U	0.366	0.373	0.365
MIT-U	0.552	0.557	0.557
SUNY.-U	0.329	0.336	0.324
PENT.-U	0.580	0.576	0.573
FOURTH-U	0.346	0.342	0.348

Table 6.18: Exponent values and the AWE heuristic. T-test with $p = 0.995$.

6.8 Heuristics vs. RL/NDP

Having discussed the usefulness of some heuristic approaches and also the difficulties present when trying to decide which heuristic is best, we now turn to the question of comparing our two separate approximation approaches. First, from the practical side, the RL/NDP methods typically require an extensive training phase, which may either be undesirable, unavailable or unneeded. However, they tend to usually lead to useful control policies, or at least the resulting policies seldom have catastrophic results. The heuristic methods are easy to compute, but have wildly varying results. For a particular domain, there is usually a heuristic which can do very well for it, but it is not always clear which heuristic to choose. There is a wide area of unexplored options for POMDPs which could combine RL/NDP techniques to possible get the best of both worlds, perhaps combining the heuristics and training a neural network to assign the proper weightings for particular situations.

Having said that, we now turn our attention to actually merging the empirical studies of these two approximation approaches. For each domain, we have selected the best RL/NDP and best heuristic results and performed a two-sided T-test to gauge whether one approach was better than the other. Table 6.19 has the results, and coincidentally they both have three domains where they are significantly better than the other.

	Best RL/NDP		Best Heuristic	
Domain	Alg.	Mean	Alg.	Mean
4X3	LIN-Q	$\boxed{1.868}$	APPROXVI	$\boxed{1.883}$
4X4	7PWLC	$\boxed{3.710}$	WE	$\boxed{3.720}$
CHEESE	LIN-Q	$\boxed{3.465}$	APPROXVI	$\boxed{3.468}$
PAINT	7PWLC	$\boxed{3.271}$	APPROXVI	$\boxed{3.250}$
SHUTTLE	LIN-Q	$\boxed{32.690}$	WE	$\boxed{32.787}$
TIGER	7PWLC	$\boxed{19.307}$	DM-MLS	$\boxed{19.843}$
NETWORK	3PWLC	$\boxed{291.343}$	APPROXVI	$\boxed{290.624}$
NONLIN	LIN-Q	$\boxed{7.158}$	APPROXVI	$\boxed{7.158}$
SACI	LIN-Q	$\boxed{14.787}$	APPROXVI	$\boxed{14.817}$
HALLWAY	7PWLC	$\boxed{1.008}$	APPROXVI	$\boxed{1.001}$
HALL.-2	7PWLC	$\boxed{0.510}$	APPROXVI	0.416
CIT	3PWLC	$\boxed{0.829}$	WE	$\boxed{0.834}$
MIT	7PWLC	$\boxed{0.868}$	AV	$\boxed{0.863}$
SUNY.	LIN-Q	$\boxed{0.771}$	MLS	0.764
PENT.	LIN-Q	$\boxed{0.819}$	WE	$\boxed{0.822}$
FOURTH	LIN-Q	$\boxed{0.596}$	WE	$\boxed{0.592}$
IFF	7PWLC	$\boxed{8.828}$	DM-MLS	8.389
BB	3PWLC	0.481	WE	$\boxed{0.668}$
MACHINE	LIN-Q	$\boxed{59.839}$	APPROXVI	$\boxed{59.884}$
ALOHA10	LIN-Q	123.871	Q-MDP	$\boxed{127.429}$
ALOHA30	LIN-Q	825.365	MLS	$\boxed{852.773}$

Table 6.19: Comparison of best heuristic and best RL/NDP variation. T-test with $p = 0.995$

6.9 Related Work

This chapter has focused on heuristic methods for approximating POMDP value functions and policies. This is far from the first research that has looked at this question and the range of techniques spans a broad space of approaches. We briefly discuss some of these approaches, though our rough characterization into topics is more for organizational purposes than it is precisely correct.

6.9.1 Grid-based

One common method to deal with continuous state spaces for MDPs includes laying a grid of points over the state space, thereby transforming the problem into a discrete problem; e.g. multi-grid techniques [17, 107]. Aside from scaling poorly with the dimensionality of the state space, these are general techniques which were not specifically developed for POMDPs and thus do nothing to exploit the shape of the value function.

Some of the earliest work in POMDPs, predating all of the exact algorithms, approached the problem by discretizing the state space [40, 54]. Although specifically geared toward partially observable environments, these methods scale miserably with the dimensionality of the state space. Lovejoy's [75] grid-based algorithm uses a more flexible scheme for establishing the grid, but still is problematic for anything but a small number of states. One of the more interesting aspects of Lovejoy's work is his methods and insights for establishing upper and lower bounds on the POMDP value function.

All of the above grid-based algorithms could be classified as *fixed-grid* methods, since the grid is established in a very regular way and never changed. These fixed-grid methods are very rigid, but allow easy interpolation for points not in the grid. Recently, Hauskrecht [46] has developed some techniques for quickly getting upper and lower bounds for the value functions, which do not depend on any particular fixed grid. The applicability to an arbitrary set of points makes his interpolation techniques especially useful for approximation schemes. Brafman [16] has also looked at applying variable grid-based methods to POMDPs, where he uses heuristic rules to decide where and when to add points to the grid.

We note that the exact algorithms of Chapter 3 can be viewed as variable-grid methods, since at each DP stage they attempt to uncover a finite number of points which will give rise to the optimal value function. Our APPROX-VI scheme is a general, though not yet precise, method for adapting these exact methods to be approximations. The work by Cheng [26] shows how his linear support algorithm can be adopted for use in a successive approximation scheme and Zhang and Liu [140] show the same for the incremental pruning algorithm.

6.9.2 Finite Memory

Another approach to approximating POMDP solutions is to only keep a finite amount of history of the process [100, 133]. This can also be viewed as a discretization of the continuous information state space space, but the discretization is in a slightly different form; now decisions are made based upon some discrete number of possible histories, instead of some interpolation from some discrete number of information points. Some of the RL/NDP techniques would also fall into the finite memory approach; for instance McCallum's RL work decides how much history is needed to do a good job predicting rewards.

6.9.3 Exploiting Structure

Although the general POMDP problem is computationally hard [95, 20, 92], there has been little work done in examining the complexity of sub-classes of POMDPs to see if certain useful restrictions could be put on the model which would make their solution tractable. White [130] shows how structure, in the form of an order on action quality, can be exploited to speed up Sondik's one-pass algorithm. A similar idea is used by Zhang [138], where he shows how to speed up the witness algorithm for problems with the structure of having relatively informative observations.

Another effort along these lines is the work by Zhang and Liu [141] which solves specific deterministically observable POMDPs as approximations to the true POMDP. By exploiting characteristics of solving these special POMDPs, more effective solution procedures can be developed and become a basis for approximations schemes.

A major limitation on the applicability of POMDPs and the techniques presented in this thesis to larger problems is in the explicit enumerative representation of all the states, actions and observations. Problems are more naturally presented in a compositional manner. For instance, the state in the robot navigation problems is more naturally thought of as consisting of two attributes, a location and an orientation, rather than a single number. For problems with more than two attributes, the compositional representation can be exponentially smaller than the enumerative scheme, which leads to the research direction of finding algorithms that can work directly on the more compact form.

Boutilier, et al [14, 15] have shown how to adapt some of the COMDP and POMDP algorithms to operate directly on the compact representation, but experience with these algorithms is severely limited. The basic idea is to represent the value functions as a tree, where the branches of the tree correspond to the different values for the state attributes. The leaves of the value function tree represent sets of states all having the same value. New leaf nodes are added only when necessary, i.e., when a new value is required for some subset of states. The unanswered questions surrounding these algorithms is whether or not useful problems have the structure necessary for the algorithms to remain compact, and what is the overhead cost in maintaining the data structures necessary in the implementation. Although these algorithms are presented as a way to arrive at optimal answers, they would seem to have the most potential as approximate algorithms where small differences between the states could be ignored.

There has been work by Dean *et al.* in solving COMDPs using compact representations [34, 35] which would seem to have natural extensions to POMDPs, though the additional complications introduced with the addition of partial observability have not yet been fully addressed.

6.9.4 Classical AI Planning

The previous discussion concerning the need for compact representations would seem strange to researchers that have been working in the area of the more traditional AI planning. In particular, we refer to classical AI planning schemes as those employing STRIPS-like operators and which derive partially ordered sequences of actions [82, 97, 98]. In these planning schemes and their derivatives, compact representation are, and always have been used, which raises the question concerning the connection between MDP algorithms and those used in these planning algorithms. This connection is elaborated in more detail in Kaelbling *et al.* [52] and here we just highlight the main points.

The basic partial order planners have actions which are deterministic, a starting state and some set of goal states which were to be achieved. With a known starting state and deterministic operators, a full policy over all possible states is not required and a *plan* or sequence of actions suffices. Although their state and action representations are compact, finding a plan requires a search through an exponential space of possibilities. Koenig [58] has shown how to recast such problems as COMDPs, which permits solution by polynomial-time algorithms. The catch here is that the conversion makes a problem that is exponential in size, since the compositional representation must now be converted into the full cross-product of all attributes. Thus the oversimplified view is that the classical AI planning does an exponential search over a polynomial representation; whereas the MDP formulation does a polynomial search over an exponential sized representation.

Discussed previously were techniques which attempt to adapt the MDP-based algorithms to compact forms; similarly, there has been much work trying to extend classical planning to handle the full generality of the MDP formulations. Although there have been many extensions to the classical planning algorithms [88, 29, 79, 136], the one that comes closest to capturing the true MDP flavor is the work on the BURIDAN [62, 61] and C-BURIDAN [37] systems, both of which allow actions with probabilistic effects and the latter which allows partial observability. However, the plans derived from the C-BURIDAN system have a limited expressibility and are not as general as possible in the full POMDP framework. Further exploration into the representational issues for classical planners is provided by Littman [73].

6.10 Conclusions

This chapter has presented an array of approximation schemes and evaluated them on a range of problems. The first major conclusion is that small POMDP problems do not pose any great difficulty for getting high quality heuristic solutions. The second result is that on a specific class of robot navigation environments, efficient heuristics can give very satisfying control policies. The RL/NDP, while applicable to these domains, requires entensive training or, when initialized with the Q-functions, do little to improve the solution. Thus, some problem have a particular structure which makes them amendable to heuristic solution. However, there are problems that do not, or for which the structure is unknown. In these cases, the RL/NDP techniques are more applicable, since they are somewhat more robust and seldom yield extremely poor answers. Finally, heuristic solutions can often give high quality control policies in other domains, though there is still room for improvements and some hybrid approach mixing heuristics with RL/NDP.

Chapter 7

Conclusions

7.1 Contributions

This thesis has contributed to advances in the exact and approximate solution of partially observable Markov decision processes. We have organized the contributions as we have organized the thesis, broken down into exact, heuristic and RL/NDP contributions.

In conjunction with Littman, Kaelbling and Zhang [23, 72, 24], this work has helped develop the witness algorithm and has helped to make improvements to the incremental pruning algorithm, both of which are currently the best exact POMDP algorithms available. Additionally, this work is the first to analyze these algorithms and their variants in terms of bounding their best and worst case complexity for the amount of effort required in their linear programming routines. It has shown that the generalized form of the incremental pruning algorithm represents an asymptotic improvement to previous algorithms in this context. We have also added some minor extensions to ideas concerning finitely transient policies which incorporate a broader class of policies with the same useful properties.

Aside from the development and theoretical analysis of these exact algorithms, this work has contributed a comprehensive implementation of the exact POMDP algorithms, as well as a comprehensive empirical comparisons of the exact algorithms using this implementation. Aside from its value to the work presented in this thesis, this implementation has proven as a useful test-bed for many researchers working on related POMDP problems and has often been the inspiration for improvements to the existing techniques.

In summary, for the exact algorithms, this thesis has contributed to the

- development of the witness algorithm, with Littman and Kaelbling;
- development of the generalized incremental pruning algorithm, with Littman and Zhang;
- detail analysis of the witness, IP, GIP and two-pass algorithms;
- various minor optimizations and improvements of the exact algorithms, with Littman;
- broadening of the class of finitely transient policies;
- implementation and empirical comparisons of all the exact algorithms.

In conjunction with Littman and Kaelbling [68], his thesis has helped derive some novel reinforcement learning rules that are applicable to POMDPs and presented some variations of previously existing techniques. We have implemented these ideas, presented some empirical comparisons using these techniques and explored some of the many possible variations available within the RL/NDP framework. The reinforcement learning contributions are the

- development of the LIN-Q and k-PWLC algorithms, with Littman and Kaelbling;
- refinements of the k-PWLC algorithms;
- implementation and empirical comparisons.

In an effort to find effective solutions to large POMDP problems, we have developed a range of heuristics which can be applied to these problems and undertook a comprehensive empirical comparison of these on a range of problems. Aside from evaluating these heuristics in synthetic simulations, along with Kurien and Kaelbling [21], we have helped to successfully apply and evaluate the POMDP model and these heuristics on a mobile robot for the purpose of navigation.

Finally, the contributions concerning the heuristics are in the

- development of various heuristic controllers for POMDPs, with Kaelbling;
- implementation and empirical comparison of the heuristics on a range of synthetic POMDP domains;
- implementation of heuristics and robot controller on autonomous robot, with Kurien.

Throughout all of the work of this thesis, many POMDP problems, spanning many domains have been developed for use in the empirical comparisons. This suite of problems has proven useful not only in this work, but by the work of many other researchers.

7.2 Future Work

Although the worst case complexity for POMDPs is somewhat disheartening, there has been little effort to try and characterize POMDPs that may be expressively restricted, but which may allow effective algorithms to be developed. This effort will require finding real POMDP problems and exploring what type of structure they may have that could be exploited. One example of an effort along these lines is some work by White [130] which exploits structure in the problem to speed up Sondik's one-pass algorithm. Insights from the structure of the problem and the nature of the algorithms could make the exact solutions of larger problems possible.

Given the computational complexity of exact algorithms, it is tempting to ignore improving the exact one-step DP algorithms for POMDPs. However, although exact VI may never be an effective method for solving realistic POMDPs, the single DP step can be an integral part of either policy iteration or some approximation algorithms. Since many of these exact algorithms may have effective approximations, improvements in the exact algorithms could lead directly to effective approximations.

There are some efforts currently in progress on policy iteration algorithms [45] using a single exact DP step. Given that PI iteration algorithms are often more effective than VI in the COMDP context, this hints that the same can be true for POMDPs. More work on improvements to PI iteration type algorithms and, more importantly, related approximate PI algorithms is the likely place where significant contributions can be made.

Even in the context of VI using the exact algorithms discussed in this thesis, we have had successes in solving problems that would otherwise be impossible by using approximate versions of these algorithms. Our method to date has mostly been heuristic and unmotivated. We would like to better characterize these approximations based upon what the exact algorithms are actually computing. This effort will lead to a better understanding of the effects of these approximations and could leads to better approximation schemes.

We have seen that there are domains where simple heuristics do very well and some where they do not, which shows much room for improvements in approximate POMDP algorithms. Aside from these heuristics being an alternative to the RL/NDP techniques, there is an interesting possibility to combine these heuristics with the RL/NDP techniques. Many successful RL/NDP efforts use hand-crafted features and let the parameter adjustment and simulations work to learn the proper function over these features. When these heuristics are viewed as features, we could use construct an RL/NDP scheme using them. More importantly, if the heuristics can capture the salient non-linearities of the environment, then simpler and more effective linear approximations could be used. There are also many interesting ways in which the heuristics could be combined using roll-out policies [11] to help these algorithms to arrive at better solutions with fewer simulations. This is one of the more promising areas for future research.

Although this and previous work has slowly been expending the sizes of problems that could be addressed with POMDP models, there are many problems that have such large state spaces, the only way to tackle them is using compositional, factored or structured models. Although there is some early work in this area [15], the effectiveness of these techniques remains largely unexplored. A related approach is to decompose the problem hierarchically, but there are many details that must first be worked.

The entire effort of this thesis makes the assumption that a full entire model, or at least a full simulation of the model, is available. Additionally, it assumes the model is unchanging over time. There are many applications of POMDPs where either or both these assumptions are not known. Either the model parameters are unknown or they may change over time. Thus, the problem becomes complicated because there is now a parameter estimation problem along with a planning problems. Some early work in learning POMDP models exist [28, 112], but more more work still needs to be done.

Another shortcoming of the POMDP approach is its limitations to discrete states. Many problems are more naturally specified as continuous space problems or have components of their states that are continuous valued. Kalman filter approaches [55, 65] to localization are the analog of the the information state update equation for POMDPs. Although this requires the state transition and observational noise to be Gaussian, Kalman filtering and the extended Kalman filter have been employed successfully and extensively in many control applications. There may be hybrid approaches that can combine the more general noise models allowed by POMDPs and those effective state estimation techniques of the Kalman filter, that permit effective heuristic solution to the control problem of problems with a mixture of discrete and continuous states.

One of the most effective methods toward making advances in research is to have challenging problems to work on. Although slowly growing, there is a need for more POMDP problems and larger POMDP problems to help spur this research effort.

Appendix A

Baseball in a Nutshell

In this appendix we attempt to briefly review the relevant portions of the game of baseball necessary for understanding the small example presented in Chapter 2. Note that we use a larger baseball domain in some of the empirical evaluations, which we discuss in Appendix H.1, but the following discussion will not be enough to completely understand that larger domain description.

Baseball's two closest relatives are the games of cricket and rounders. The overall scenario is for one team to attempt to hit a ball (using some form of long stick) which is thrown by the opposing team. The team that throws the ball is attempting to have the batter either miss striking the ball entirely, or to have them hit the ball weakly. The teams also take turns in the two aspects of throwing and hitting and the team that performs better than the other at hitting the ball is the winner.

In the game of baseball, the person throwing the ball (called a *baseball*) is referred to as the *pitcher*, whereas the person trying to hit the ball, uses a *bat*, and is called a *batter*. The pitcher has a team of *fielders* around that can catch the batted balls, and if the balls were hit weakly enough, the fielders can make a play such that the batter's attempt is deemed unsuccessful. When the batter's attempt is unsuccessful, it is called making an *out*, while a successful batter is credited with making a *hit*. After a certain number of hits, the batter and his team are credited with points or *runs* in baseball. After a certain number of outs, the pitcher and his team get a chance at batting, while the batter and his team go into the *field* with one of those players taking on the role of the pitcher.

One series of being the team at bat and then being the team in the field is called an *inning*. Normally a baseball game lasts 9 innings and the team scoring the most runs at the end of this time is declared the winner. In case of ties, a sequence of full innings are played until at the end of one of these innings one team has gained the advantage over the other; i.e., more runs are credited.

Roughly, the more hits a team gets, the more runs they score and the more likely they are to win the game. Likewise, the less hits a team gives up, the less runs the opponents score and the more likely they are to win. Thus the main tension in the game is between the pitcher and the batter.

Aside from the players in the game, there is a *manager* who is responsible for deciding what order his players should bat, who should pitch, and many other decisions involving details of the game not discussed here. In baseball, once a pitcher is removed from the game, they can no longer pitch in that game. Thus, the manager must carefully decide when and when not to take a pitcher out of the game. In baseball, pitching is a specialty task that not all players on the team are competent enough to do. Although a baseball team typically is comprised of 25 or so players, only about 10 of them are usually good enough at pitching that the manager would decide to let them pitch. The pitchers not currently pitching are referred to collectively as the team's *bull-pen* for archaic reasons.

Pitching, like any other athletic activity, is subject to complex physical and mental interactions, which means that the ability and performance level of a pitcher is subject to fluctuations. The manager would like to get as much out of each pitcher as possible, but also wants to recognize when a pitcher might be having a bad day and remove him/her before the other team accumulates too many hits.

The hidden state in our example is the combined physical and mental condition of the pitcher. This state is often hidden from the pitchers themselves, since the criteria they may use to assess their own condition may not reflect the criteria necessary for performing well as a pitcher. Even when a pitcher knows their own physical or mental condition to be below the normal levels, the pitcher can be reluctant to inform the manager for any of a host of complicated reasons; e.g., ego, contract status, embarrassment, etc. The manager is faced with the task of determining the condition of the pitcher with only limited information.

In our example, the pitcher's performance against each batter provides the manager with evidence of the internal condition of the pitcher. Based upon this evidence, which we break down into the simple cases of the batter making and out or getting a hit, the manager must decide after each batter whether to let the current pitcher continue, or to replace him with one of the pitchers in the bull-pen.

Aside from the initial conditions of a pitcher on a given day, the condition of a pitcher can deteriorate as the game progresses. The physical act of pitching a baseball requires strenuous activity, which even the most fit of people can only effectively perform for a limited amount of time. Thus, at any given point in time, the pitcher can become fatigued, causing performance to suffer. Although we have modeled the probability of a pitcher becoming fatigued as constant over all time, a realistic model would have this probability be a function of time; i.e., a non-stationary state transition function.

Appendix B

PWLC Properties

There are a number of properties of PWLC functions and operations on their representations which are used throughout this thesis. This appendix summarizes these useful properties. For this appendix, we will use V^A and V^B to represent two PWLC value functions over information space and let A and B, respectively, be the two sets of vectors representing those value functions where

$$V^A(b) = \max_{\alpha \in A} b \cdot \alpha$$
$$V^B(b) = \max_{\beta \in B} b \cdot \beta \quad .$$

We define an equivalency relation between the representation and the value function, which notationally is $A \equiv V^A$ and $B \equiv V^B$.

Repeating Propositions 2.3.1 and 2.3.2 we have

- $V^A + V^B$ is a PWLC function, and
- $\max(V^A, V^B)$ is a PWLC function.

B.1 Cross-sum

In this section we formally define the cross-sum operator, $\oplus$, and later will discuss some of its properties when applied to PWLC functions represented by a set of vectors.

Definition B.1.1 *The cross-sum operation on two sets of vectors, A and B is*

$$A \oplus B = \{\alpha + \beta | \, \alpha \in A, \beta \in B\} \quad .$$

As a direct result of the properties of the addition operator, this operator is commutative and associative.

Definition B.1.2 *Similar to the notation, $\sum$, for addition, we define*

$$\bigoplus_{i=1}^{N} A_i = A_1 \oplus A_2 \oplus \ldots \oplus A_N \quad .$$

B.2 Representation Properties

Relating the union and cross-sum operations on the representations we have that

$$A \oplus B \equiv V^A + V^B$$

and

$$A \cup B \equiv \max(V^A, V^B) \quad ,$$

and we see that the union or cross-sum of two sets of vectors are themselves representations for a PWLC function.

As discussed in Section 3.1.1 concerning parsimonious sets, a given value function can have many different representations as a set of vectors. The pruning operator, whose semantics are given by the PRUNE routine of Table 3.4 in Section 3.1.1, operates on a set of vectors to produce another set of vectors. The PRUNE operation was introduced as a way to convert a representation of a PWLC value function to a unique minimal set. Because of this we have, $\forall b \in \mathcal{B}$,

$$\begin{aligned} V^A(b) &= \max_{\alpha \in A} b \cdot \alpha \\ &= \max_{\alpha \in \text{PRUNE}(A)} b \cdot \alpha \end{aligned}$$

and, with some abuse of notation,

$$A \equiv \text{PRUNE}(A) \equiv V^A$$

where the equivalency relation is extended to incorporate a relation between two sets.

This gives us the properties

$$\begin{aligned} \text{PRUNE}(A \cup B) &\equiv \max(V^A, V^B) \\ \text{PRUNE}(A \oplus B) &\equiv V^A + V^B \quad , \end{aligned}$$

which lead directly to

$$\begin{aligned} \text{PRUNE}(A \cup B) &\equiv \text{PRUNE}(\text{PRUNE}(A) \cup \text{PRUNE}(B)) \\ \text{PRUNE}(A \oplus B) &\equiv \text{PRUNE}(\text{PRUNE}(A) \oplus \text{PRUNE}(B)) \quad . \end{aligned}$$

Appendix C

Random Distributions

To problem we address here is how to generate a random discrete probability distribution, p such that the entire probability space, an N-dimensional simplex, is sampled in a uniform manner. The obvious approach would be to first generate N real numbers, $p(1), p(2) \ldots p(N)$, each being drawn from a uniform distribution on the interval $[0, 1]$ and then normalize this vector so that

$$p := \frac{p}{\sum_{i=1}^{N} p(i)} \quad ,$$

to satisyfy the simplex constraints. This generates a probability distribution, but does not generate distributions uniformly randomly over the probability space. This algorithm skews the distribution in such a way that points close to the simplex corners and borders are much less likely than points on the interior, or closer to the uniform distribution. Figure C.1 shows $10,000$ points generated on an $N = 3$ simplex. Note that the simplex constraints force a two-dimensional space.

The correct algorithm for ensuring distributions are chosen uniformly

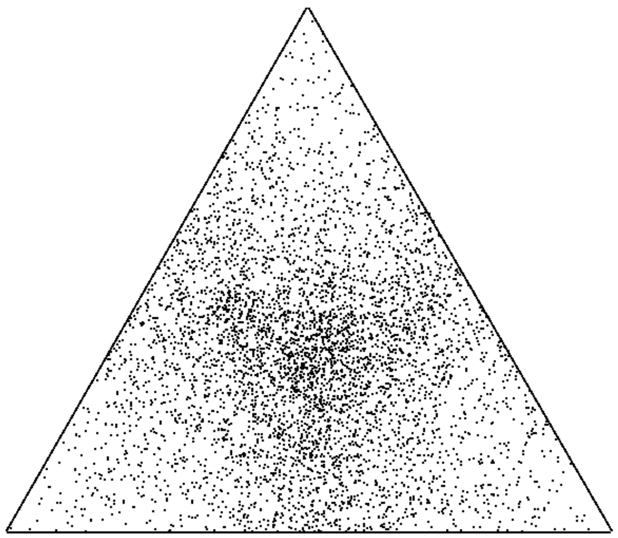


Figure C.1: Random probability points generated according to a naive algorithm.

at random is given in Table C.1[1] In this algorithm, the function `rand()` is simply a routine that returns a uniformly random real number on the interval $[0,1]$. Figure C.2 shows $10,000$ points generated according to this algorithm.

[1]Thanks to John Hughes for showing us this algorithm.

```
randomDistribution()
    p(1) := 1
    for each i ∈ {2, 3, ..., N}
        p(i) := 1.0 − ⁱ√rand()
        for each j ∈ {1, ..., j}
            p(j) := 1.0 − p(i)
        end for each j
    end for each i
    return p
end randomDistribution
```

Table C.1: Routine for generating a uniformly random discrete probability distribution.

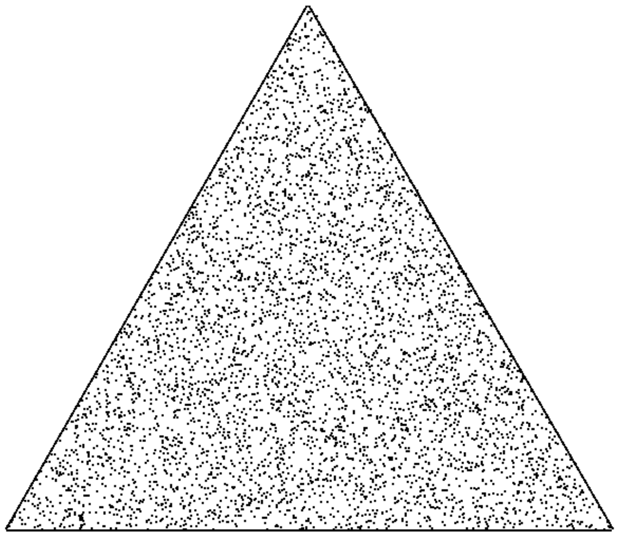


Figure C.2: Random probability points generated according to the correct algorithm.

Appendix D

Finitely Transient Policies

For a stationary policy of an infinite horizon POMDP problem, there is a property called *finite transience* (f.t.) which was introduced by Sondik [117]. The interest in f.t. policies lies in Sondik's theorem that if an infinite horizon policy is f.t., then its value function is p.w.l. and can be computed relatively easily by solving a system of equations. However, the requirements that Sondik defines for f.t. policies are much stronger than are needed to ensure a p.w.l. cost function for a policy as we show here. In this section we will discuss the finitely transient property and introduce a more relaxed criterion that will have the same nice properties as the f.t. policies. Before defining the finitely transient property and our extension, we require some additional concepts, which are due to Sondik.

We will use a POMDP problem with 2 states, 3 actions and 2 observations, whose parameters are given in Table D.1[1]. For this problem, the discount factor will be $\rho = 0.95$ and we will be considering the policy shown in Figure D.1. Since this is a two state problem, an information state can be

[1]This problem has appeared elsewhere and is known as the "tiger" problem. [23]

$\tau(\cdot, 0, \cdot)$	s	s'
s	1.00	0.00
s'	0.00	1.00

$\tau(\cdot, \{0,1\}, \cdot)$	s	s'
s	0.50	0.50
s'	0.50	0.50

$o(0, \cdot, \cdot)$	z	z'
s	0.85	0.15
s'	0.15	0.85

$o(\{1,2\}, \cdot, \cdot)$	z	z'
s	0.50	0.50
s'	0.50	0.50

$r(\cdot, a)$	a		
	0	1	2
s	-1	-100	10
s'	-1	10	-100

Table D.1: Model parameters for f.t. example.

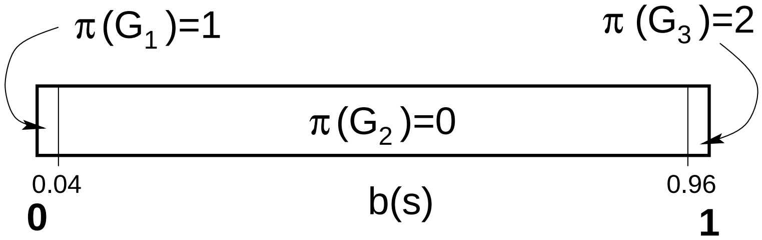


Figure D.1: Policy regions for f.t. example.

represented with a single number, namely $b(s)$ since $b(s') = 1 - b(s)$. This policy imposes a partition, $G^\pi = \{G_1^\pi, G_2^\pi, G_3^\pi\}$, on the information space consisting of three regions. For this example, the precise partition, which can be represented as a set of intervals over $b(s)$, is

$$G_\pi = \left\{ \begin{array}{lll} G_1^\pi = [& 0.0000000000 & 0.0396544425 \quad], \\ G_2^\pi = [& 0.0396544425 & 0.9603455575 \quad], \\ G_3^\pi = [& 0.9603455575 & 1.0000000000 \quad] \end{array} \right\} .$$

Since we define a single action over each partition element, we will use the notation $\pi(G_i) = \pi(i)$ for the action assigned to partition element G_i. For the example policy we have: $\pi(1) = 1$, $\pi(2) = 0$ and $\pi(3) = 2$.

It is of interest to be able to easily find the value of an arbitrary policy defined over information space, e.g., the value determination step of policy iteration. We will make the simplifying assumptions that the policy is specified with a finite number of connected regions, where each region is either convex or made up of a finite number of convex regions.

Let D_π be the set of information points at which a policy is discontinuous. For the example, as shown in Figure D.1, policy consists only of three regions with discontinuity set

$$D_\pi = \{0.0396544425, 0.9603455575\} \quad ,$$

where we assume that the policy is continuous at the simplex corners. Again, with two states we can represent the regions borders with a single number. Note that D_π is a set of points and not a set of intervals. Also for $|\mathcal{S}| > 2$ all of the results of this section still apply, but the representation of the partitions and discontinuities must be done using a more complex system of hyper-planes instead of simple intervals and points. Sondik [117] shows how this is done for $|\mathcal{S}| > 2$.

Let A be a subset of the information state space, $\mathcal{B}$, and let the information transformation function on a set of points be defined with

$$T(A, a, z) = \{b_z^a | b \in A\} \quad ,$$

and note that $T(\{b\}, a, z)$ is equivalent to the one element set $\{b_z^a\}$. The $T()$ function simply converts one set of points into another.

Definition D.0.1 *The set of all possible transformed information states of* *a set A is*

$$T_\pi(A) = \{T(b, \pi(b), z) | \forall b \in A, \forall z \in \mathcal{Z}\} \quad .$$

Let $S_\pi^0 = \mathcal{B}$ and define

$$S_\pi^n = T_\pi(S_\pi^{n-1}) \quad , n \geq 1 \quad .$$

This states that S_π^n is the set of all possible information states that the process could have after following the policy π for n steps. This makes no assumptions about the initial information state, or the sequences of observations received.

In the example policy, with $S_\pi^0 = \mathcal{B}$, S_π^1 consists of all information states in the interval

$$\left[\, 0.0072340834 \; 0.9927659166 \,\right] \quad .$$

Further, we see that $S_\pi^2 = S_\pi^1$, which means that $S_\pi^n = S_\pi^1$ for all $n > 1$.

Sondik gives the following definition on page 71 of his thesis:

Definition D.0.2 *A stationary policy π is* finitely transient *if and only if there is an integer $n < \infty$ such that*

$$D_\pi \cap S_\pi^n = \emptyset \quad .$$

The smallest such n is called the index *of the finitely transient policy and is labeled n_π.*

By this definition, our example policy is not finitely transient since both policy discontinuity points lie in the interval defined by S_π^n for all n. The transition dynamics of this example are such that there is a non-zero probability of being at a discontinuity at any step, $n < \infty$, in the future. This property removes this policy from the class of f.t. policies, though we shall see that, despite this, this policy has the same nice properties as f.t. policies.

There is another way to approach f.t. policies (also due to Sondik) that uses a sequence of sets derived from the discontinuities of the policy. This looks at the inverse problem, and attempts to find all possible states which could reach the policy's discontinuities. We define the inverse information transformation on a set of information points as

$$T_{\pi}^{-1}(A) = \left\{ b | b_{z}^{\pi(b)} \in A, \text{ for some } z \in \mathcal{Z} \right\} \quad , n \geq 0 \quad .$$

Letting $D^0 = D_{\pi}$, which is just the set of discontinuities of the policy, we then define

$$D^{n+1} = T_{\pi}^{-1}(D^n) \quad , \forall n \geq 0 \quad ,$$

which defines the set of all information states which can reach a discontinuity of the policy in $n+1$ steps following the policy π. We can find these sets by using the inverse of the information state transformation function and Sondik's construction methods [117], though we must handle the special case of a non-invertible transformation function explicitly.

For actions 1 and 2 of our example, the information state transformation is non-invertible, since no matter what the information state is, the result of these actions is the information state [0.5 0.5]. Since this point is not one of the discontinuities of the policy, those two actions will not contribute to any of the D^n sets, unless we find $[\ 0.5\ 0.5\ ] \in D^{n-1}$, which does not occur in this example.

Looking at our example policy the sequence of discontinuity sets is

$$D^0 = \{0.0396544425, 0.9603455575\}$$

$$D^1 = \{0.1896187811, 0.8103812189\}$$

$$D^2 = \{0.4299360872, 0.5700639128\}$$

$$D^3 = \{0.1174593043, 0.1896187811, 0.8103812189, 0.8825406957\}$$

$$D^4 = \{0.4299360872, 0.5700639129\} \quad .$$

At this point we see that $D^4 = D^2$ which means that the sequence will repeat indefinitely from that point.

Sondik gives the following Lemma

Lemma D.0.1 *D^n is the first empty set in the sequence D^1, D^2, ... if and only if the policy π is finitely transient with $n_\pi = n$.*

This Lemma follows almost immediately from the definition of a f.t. policy. Using the f.t. property, Sondik goes on to show that these f.t. policies have some desirable properties. However, we see again, that the example's policy does not satisfy this criterion and is not f.t. It turns out that this example policy has the same nice properties as f.t. policies, and so we would like a less stringent criterion that ensures we get those properties.

Define $\overline{D}^n = \bigcup_{i=0}^{n} D^i$, which is simply every point that leads to a discontinuity of the policy in n or fewer steps.

Definition D.0.3 *If* $\overline{D}^{n+1} = \overline{D}^n$ *for any* $n < \infty$ *we say the policy is* extended finitely transient *(e.f.t.) with index* n_π*.*

Theorem D.0.1 *If* $\overline{D}^{n+1} = \overline{D}^n$*, then* $\overline{D}^m = \overline{D}^n$ *for all* $m > n+1$*.*

Proof We prove this by contradiction and note that the construction process ensures that $\overline{D}^i \subseteq \overline{D}^j$ for all $j > i$. Assume $\overline{D}^m$ is the first set in the sequence that differs from $\overline{D}^n$. Then there must be a point $b \in \overline{D}^m$, where $b \notin \overline{D}^n$, such that for some observation z, $b_z^{\pi(b)} \in \overline{D}^{m-1}$. Since this is the first discontinuity set that differs from $\overline{D}^n$, we know that $\overline{D}^{m-1} = \overline{D}^{n+1} = \overline{D}^n$ and we must have $b_z^{\pi(b)} \in \overline{D}^n$. Since $b \in T^{-1}(\{b_z^{\pi(b)}\})$, b must be in $\overline{D}^{n+1}$ and we have a contradiction because we defined $b \notin \overline{D}^n$ and $\overline{D}^{n+1} = \overline{D}^n$. ■

For our example, the sequence we get is

$$\overline{D}^0 = \{0.0396544425, 0.9603455575\}$$

$$\begin{aligned}\overline{D}^1 = \{&0.0396544425, 0.1896187811, 0.8103812189,\\ &0.9603455575\}\end{aligned}$$

$$\begin{aligned}\overline{D}^2 = \{&0.0396544425, 0.1896187811, 0.4299360872,\\ &0.5700639128, 0.8103812189, 0.9603455575\}\end{aligned}$$

$$\begin{aligned}\overline{D}^3 = \{&0.0396544425, 0.1174593043, 0.1896187811,\\ &0.4299360872, 0.5700639128, 0.8103812189,\\ &0.8825406957, 0.9603455575\}\end{aligned}$$

$$\overline{D}^4 = \{0.0396544425, 0.1174593043, 0.1896187811,$$
$$0.4299360872, 0.5700639128, 0.8103812189,$$
$$0.8825406957, 0.9603455575\} \ .$$

We see that our example policy does have the e.f.t. property, with index $n_\pi = 3$. Notice that the e.f.t. property is implied by the f.t. property since $D^n = \emptyset$ ensures that $\overline{D}^n = \overline{D}^{n-1}$. We will now closely follow Sondik's development for the properties of f.t. policies, except we will extend this to the e.f.t. case.

For Sondik, his f.t. property assures that, regardless of the initial information state, after a finite number of steps it will be impossible to be at an information state which lies on a border of the policy partition. For e.f.t. policies, there may be no finite number of steps in which we can give this same guarantee. However, we will be able to guarantee that, for some finite number of steps, we only reach a partition boundary point if the initial information state is itself a partition boundary point.

We define a partition $G^n = \{G_i^n\}$ by using the set $\overline{D}^n$ as the defining borders for the partition regions. Since $\overline{D}^{n-1} \subseteq \overline{D}^n$, each partition derived from the discontinuity set sequence is a refinement of the previous partition with the final partition G^{n_π}, which can not be refined further since $\overline{D}^{n+1} = \overline{D}^n$ for all $n \geq n_\pi$. Since we are mainly concerned with the final partition, we will drop the superscript such that $G^{n_\pi} = G$ and $G_i^{n_\pi} = G_i$.

For our example policy we have

$$G^0 \equiv \left\{ \begin{array}{c} [\ 0.0000000000 \quad 0.0396544425\ ] \\ [\ 0.0396544425 \quad 0.9603455575\ ] \\ [\ 0.9603455575 \quad 1.0000000000\ ] \end{array} \right\}$$

$$G^1 \equiv \left\{ \begin{array}{c} \left[\begin{array}{cc} 0.0000000000 & 0.0396544425 \end{array} \right] \\ \left[\begin{array}{cc} 0.0396544425 & 0.1896187811 \end{array} \right] \\ \left[\begin{array}{cc} 0.1896187811 & 0.8103812189 \end{array} \right] \\ \left[\begin{array}{cc} 0.8103812189 & 0.9603455575 \end{array} \right] \\ \left[\begin{array}{cc} 0.9603455575 & 1.0000000000 \end{array} \right] \end{array} \right\}$$

$$G^2 \equiv \left\{ \begin{array}{c} \left[\begin{array}{cc} 0.0000000000 & 0.0396544425 \end{array} \right] \\ \left[\begin{array}{cc} 0.0396544425 & 0.1896187811 \end{array} \right] \\ \left[\begin{array}{cc} 0.1896187811 & 0.4299360872 \end{array} \right] \\ \left[\begin{array}{cc} 0.4299360872 & 0.5700639128 \end{array} \right] \\ \left[\begin{array}{cc} 0.5700639128 & 0.8103812189 \end{array} \right] \\ \left[\begin{array}{cc} 0.8103812189 & 0.9603455575 \end{array} \right] \\ \left[\begin{array}{cc} 0.9603455575 & 1.0000000000 \end{array} \right] \end{array} \right\}$$

$$G^3 \equiv \left\{ \begin{array}{c} \left[\begin{array}{cc} 0.0000000000 & 0.0396544425 \end{array} \right] \\ \left[\begin{array}{cc} 0.0396544425 & 0.1174593043 \end{array} \right] \\ \left[\begin{array}{cc} 0.1174593043 & 0.1896187811 \end{array} \right] \\ \left[\begin{array}{cc} 0.1896187811 & 0.4299360872 \end{array} \right] \\ \left[\begin{array}{cc} 0.4299360872 & 0.5700639128 \end{array} \right] \\ \left[\begin{array}{cc} 0.5700639128 & 0.8103812189 \end{array} \right] \\ \left[\begin{array}{cc} 0.8103812189 & 0.8825406957 \end{array} \right] \\ \left[\begin{array}{cc} 0.8825406957 & 0.9603455575 \end{array} \right] \\ \left[\begin{array}{cc} 0.9603455575 & 1.0000000000 \end{array} \right] \end{array} \right\}.$$

Because this policy is e.f.t. with index 4, we know that $G^i = G^3$ for all $i \geq 4$ and we have

$$G \equiv \left\{ \begin{array}{rl} G_1 = & \left[\begin{array}{cc} 0.0000000000 & 0.0396544425 \end{array} \right] \\ G_2 = & \left[\begin{array}{cc} 0.0396544425 & 0.1174593043 \end{array} \right] \\ G_3 = & \left[\begin{array}{cc} 0.1174593043 & 0.1896187811 \end{array} \right] \\ G_4 = & \left[\begin{array}{cc} 0.1896187811 & 0.4299360872 \end{array} \right] \\ G_5 = & \left[\begin{array}{cc} 0.4299360872 & 0.5700639128 \end{array} \right] \\ G_6 = & \left[\begin{array}{cc} 0.5700639128 & 0.8103812189 \end{array} \right] \\ G_7 = & \left[\begin{array}{cc} 0.8103812189 & 0.8825406957 \end{array} \right] \\ G_8 = & \left[\begin{array}{cc} 0.8825406957 & 0.9603455575 \end{array} \right] \\ G_9 = & \left[\begin{array}{cc} 0.9603455575 & 1.0000000000 \end{array} \right] \end{array} \right\},$$

which is the final partition and is shown in Figure D.2.

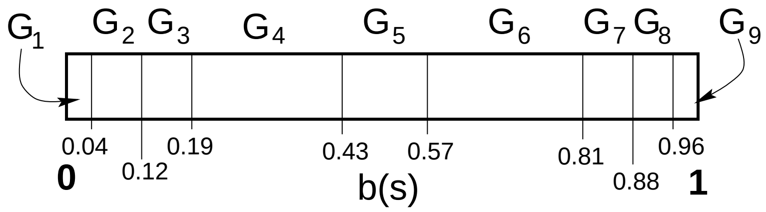


Figure D.2: Final constructed partition for f.t. example.

Lemma D.0.2 *All information states in a partition element G_i^n are assigned the same action, $\pi(i)$, by π.*

Proof Since D_π is a subset of the boundaries of G_i^n, the partition must be a refinement of G^π. Since the policy partition initially assigns only one action to each partition element, it must assign one and only one action to each partition element G_i^n. ∎

Lemma D.0.3 *For the final partition formed by a e.f.t. policy, if $b \in G_i$ and $b_z^{\pi(i)} \in G_j$, then $\forall \widehat{b} \in G_i$ we must have $\widehat{b}_z^{\pi(i)} \in G_j^n$.*

Proof Let two distinct points, b and $\widehat{b}$ lie within the region of partition element G_i where $b_z^{\pi(i)} \in G_j$ and $\widehat{b}_z^{\pi(i)} \in G_k$ for some observation z. We now assume that $G_j \neq G_k$ and proceed to find a contradiction. Let l be the line segment that lies between b and $\widehat{b}$. Since we only allow convex regions, l must lie entirely within the interior of the region of G_i.

Thus, the line segment $T(l, \pi(i), z)$ will have endpoints in G_j and G_k. By our assumption $G_j \neq G_k$ and the properties of the transformation function, the line segment must cross at least one region boundary, and there must be a point $b^* \in l$ such that $b_z^{*,\pi(i)}$ lies on the boundary of two regions. However,

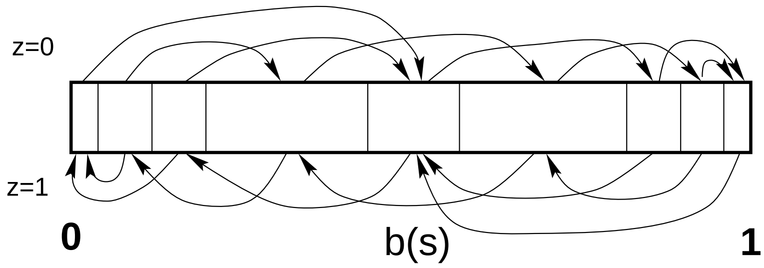


Figure D.3: Information state transitions on partition for e.f.t. example.

if $b_z^{*,\pi(i)}$ is on the boundary, then $b_z^{*,\pi(i)} \in \overline{D}^{n_\pi}$ and there must be some set D^n that contains this point. By nature of construction, b^* must be in the set D^{n+1}. If $b^* \in D^{n+1}$, then it must also be in $\overline{D}^{n_\pi}$. However, b and $\widehat{b}$ were chosen to be in the partitions element's interior and the convexity of the regions means that all points in l must also be in the interior, which is the contradiction since we find b^* in l and in $\overline{D}^{n_\pi}$. ■

Definition D.0.4 *When $b \in G_i^n$ and $b_z^{\pi(i)} \in G_j^n$ then $\nu(i,z) = j$.*

Thus, given a partition derived from an e.f.t. policy, we can easily define this $\nu(\cdot,\cdot)$ mapping by selecting a point in each region and transforming it for each observation. From Lemma D.0.3, we see it does not matter which information state we select for each partition element. Returning to our example, we can construct the mapping shown in Table D.2 and illustrated in Figure D.3.

The following lemma appears, and is proved in Sondik's thesis [117] as Lemma 3.4 on page 72. It shows the form of the value function for any stationary policy evaluated over the infinite horizon. Note that this theorem does not say that all policies have p.w.l. value functions. However, below we

i	z	z'
1	5	5
2	4	1
3	5	1
4	6	2
5	7	3
6	8	4
7	9	5
8	9	6
9	5	5

Table D.2: Partition transition function $\nu(\cdot,\cdot)$ for the e.f.t. example.

will use this lemma to prove that e.f.t. policies do have p.w.l. value functions.

Lemma D.0.4 *The value function, $V_\pi(b)$, of a policy π can be written*

$$V_\pi(b) = b \cdot g(b|\pi)$$

where $g(b|\pi)$ is an $|\mathcal{S}|$-vector that is the unique bounded solution to the vector equation

$$g(b|\pi) = r(\pi(b)) + \rho \sum_z P^{\pi(b),z} g(b_z^{\pi(b)}|\pi) \quad . \tag{D.1}$$

Note that this lemma does not assume that there are a finite number of $g(b|\pi)$ which satisfy this equation.

The following theorem just extends Sondik's Thesis Theorem 3.4 to the case of e.f.t. policies. The proof is exactly the same as presented by Sondik for the f.t. case, since his proof relies only upon properties of f.t. policies which we have shown to exist for e.f.t. policies.

Theorem D.0.2 *If a policy π is e.f.t. then $V_\pi(\cdot)$ is p.w.l.*

Proof By Lemma D.0.4, the value function can be written as $V_\pi(b) = b \cdot g(b|\pi)$ where $g(b|\pi)$ is the unique solution to Equation D.1. The only thing remaining to be proven is that there are a finite number of $g(b|\pi)$ vectors.

Since the policy is e.f.t.,

- we can construct a finite sized partition set $G = \{G_i\}$ using $\overline{D}^{n_\pi}$ as the region boundaries by Theorem D.0.1,
- the same action is defined over all information points in a given partition element by Lemma D.0.2,
- a $\nu(\cdot, \cdot)$ mapping exists by Lemma D.0.3.

If we assume that for each partition element and for all $b \in G_i$ we have $g(b|\pi) = g_i$. Since for all $\hat{b} \in G_i$, $\pi(b) = \pi(i)$, Equation D.1 can be transformed into a finite set of equations

$$g_i = r(\pi(i)) + \rho \sum_z P^{\pi(i),z} g_{\nu(i,z)} \quad , \tag{D.2}$$

which must have a unique solution by the properties of right-hand side[2].

Since Equation D.1 and Equation D.2 are of the same form and have unique solutions, they must be one and the same solution. Thus, the solution to Equation D.2 must be the policy's value function, which shows it has a finite number of segments. ■

[2]The right-hand side is a vector contraction mapping for the function $g(\cdot|\pi)$ under the vector supremum norm. See the proofs of Lemma 3.1 and Lemma 3.4 of Sondik's thesis which discuss this further.

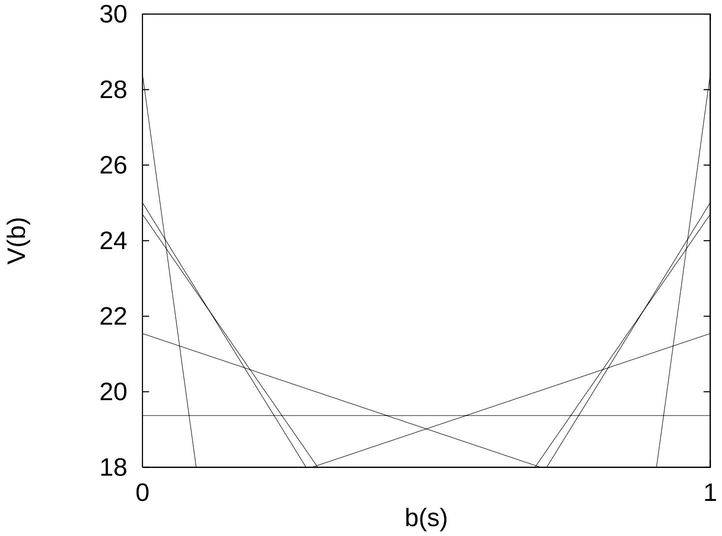


Figure D.4: Optimal value function for f.t. example.

For our example, we can set up the system of equations from the $\nu(\cdot,\cdot)$ function and solve to get the p.w.l. value function represented by

$$V_\pi(\cdot) \equiv \left\{ \begin{array}{lrrl} g_1 = [& -81.597200063 & 28.402799937 &] \\ g_2 = [& 0.690888139 & 25.004972735 &] \\ g_3 = [& 3.014778938 & 24.695680939 &] \\ g_4 = [& 16.493485015 & 21.541837097 &] \\ g_5 = [& 19.371368356 & 19.371368356 &] \\ g_6 = [& 21.541837097 & 16.493485015 &] \\ g_7 = [& 24.695680939 & 3.014778938 &] \\ g_8 = [& 25.004972735 & 0.690888139 &] \\ g_9 = [& 28.402799937 & -81.597200063 &] \end{array} \right\} ,$$

where this value function is shown in Figure D.4. Note that this value function is both p.w.l. and convex, though convexity of a policy's value function is not always guaranteed. In this case, the example policy we used corresponded to the optimal policy, which explains its convexity.

Sondik makes the following conjecture about f.t. policies:

Conjecture D.0.1 *A policy with a p.w.l. value function is f.t.*

and shows that it is false by way of a counter-example. Thus, the same question arises in the context of e.f.t. policies:

Conjecture D.0.2 *A policy with a p.w.l. value function is e.f.t.*

The natural place to start is with Sondik's counter-example which shows that the dynamics of the information state transformation makes the information states asymptotically approach a single point for a particular combination of an action and an observation. Since this limiting point lies within the action's region, once an information state transforms into the region, for a the particular observation the information state will approach the limiting point, but only reach it in the limit. Since this limiting point lies within the region, the S^n sequence will never be empty. In fact, Sondik shows that the set of reachable states has $\lim_{n\to\infty} S^n_\pi = [\ 0.075807\ 0.763158\ ]$[3], which includes the policy discontinuity point 0.6. Since he defines f.t. policies as $D_\pi \cap S^n_\pi = \emptyset$, this policy is clearly not finitely transient.

However, we have defined e.f.t. policies using the sequence of discontinuity sets $\overline{D}^n_\pi$. To show a policy is not e.f.t. we have to show $\lim_{n\to\infty} \overline{D}^n_\pi \neq \emptyset$. We have not yet shown a counter-example to Conjecture D.0.2, but suspect there are many.

[3]Sondik states that S^∞ is $[\ 0.076\ 0.766\ ]$.

Appendix E

Neighbor Properties

There are some properties of the neighbor relation from Section 3.2.1 which may not be immediately obvious. We use this appendix to briefly highlight a few. We use a simple POMDP where we have only two observations and two states. Thus we want to explore the relationship between a vector and its neighbors in terms of how Γ_n^a is constructed from the individual $\Gamma_n^{a,z}$ sets.

Consider the case of a vector γ which has a non-empty region in Γ_n^a. One obvious fact is that for a given neighbor, ν, it may or may not be the case that $R(\nu, \Gamma_n^a) = \emptyset$.

The first somewhat counter-intuitive property is the fallacy that for a neighbor with a non-empty region in Γ_n^a, that $R(\nu, \Gamma_n^a)$ and $R(\gamma, \Gamma_n^a)$ must be adjacent. Figure E.1 shows a counter-example where a neighbor's region is non-adjacent.

Another not so obvious fallacy is that if two regions $R(\gamma, \Gamma_n^a)$ and $R(\gamma', \Gamma_n^a)$ are adjacent, then $\gamma' \in \mathcal{N}(\gamma)$. Figure E.2 is a counter-example. This figure also shows another non-obvious case where, for all neighbors ν of a vector γ, we have $R(\nu, \Gamma_n^a) = \emptyset$.

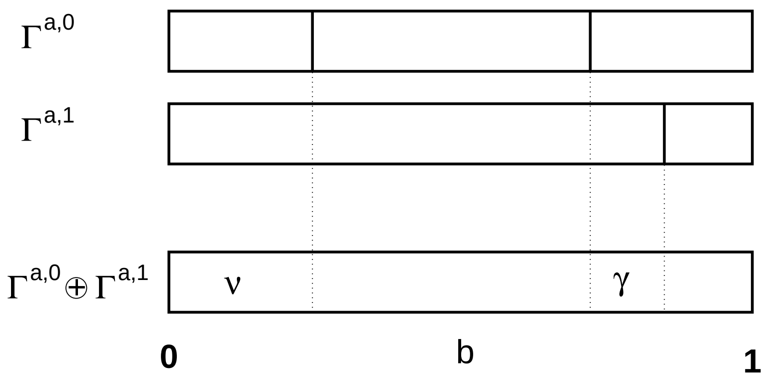


Figure E.1: A vector's neighbor with a non-adjacent region.

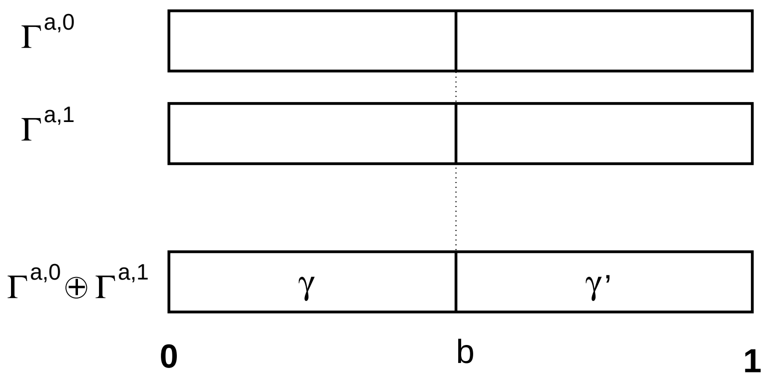


Figure E.2: A situation where adjacent regions are not neighbors and where all neighbors have empty regions.

Appendix F

Full DP Example

In this appendix, we show how the witness and incremental pruning algorithms work on a simple example for a few value iteration steps. This is intended to serve two purposes; to concretely illustrate the operation of the two algorithms; and to provide a point of reference for researchers that may attempt to implement these or related algorithms.

We use the the simple 2-state, 2-action, 2-observation POMDP baseball example from Chapter 2 where the model parameters are given in Tables 2.1, 2.6 and 2.7 and the discount factor is $\rho = 0.95$. We assume that the terminal rewards are all zero so that $\Gamma_0 = \{[\ 0.0000\ 0.0000\ ]\}$. We will use the following indices to make the notation succinct: action `pitch` $= 0$; action `bull-pen` $= 1$; observation `out` $= 0$; observation `hit` $= 1$. In addition, the first component of the vectors shown corresponds to state $s =$ `good` and the second component is $s' =$ `bad`. We show the first 3 DP steps using the incremental pruning algorithm and the 4^{th} step using the witness algorithm.

F.1 Incremental Pruning

We start with the no cost terminal value function

$$\Gamma_0 = \left\{ \begin{bmatrix} -0.00000 & -0.00000 \end{bmatrix} \right\}$$

and generate the $\Gamma_1^{0,z}$ sets using Equation 3.2 to get

$$\Gamma_1^{0,0} = \left\{ \begin{bmatrix} 0.03250 & -0.46250 \end{bmatrix} \right\}$$
$$\Gamma_1^{0,1} = \left\{ \begin{bmatrix} 0.03250 & -0.46250 \end{bmatrix} \right\} .$$

Since there is only a single vector in each set, the full single action value function is

$$\Gamma_1^0 = \left\{ \begin{bmatrix} 0.06500 & -0.92500 \end{bmatrix} \right\} ,$$

which simply corresponds to the immediate rewards for action 0. By the same procedure, we derive the other action's value function as its immediate rewards and get

$$\Gamma_1^1 = \left\{ \begin{bmatrix} -0.3750 & -0.3750 \end{bmatrix} \right\} .$$

Combining the two action value functions, we find that each vector has a non-empty region, so the final value function is represented with

$$\Gamma_1 = \left\{ \begin{array}{c} \begin{bmatrix} 0.06500 & -0.92500 \end{bmatrix} \\ \begin{bmatrix} -0.37500 & -0.37500 \end{bmatrix} \end{array} \right\} ,$$

which is shown in Figure F.1.

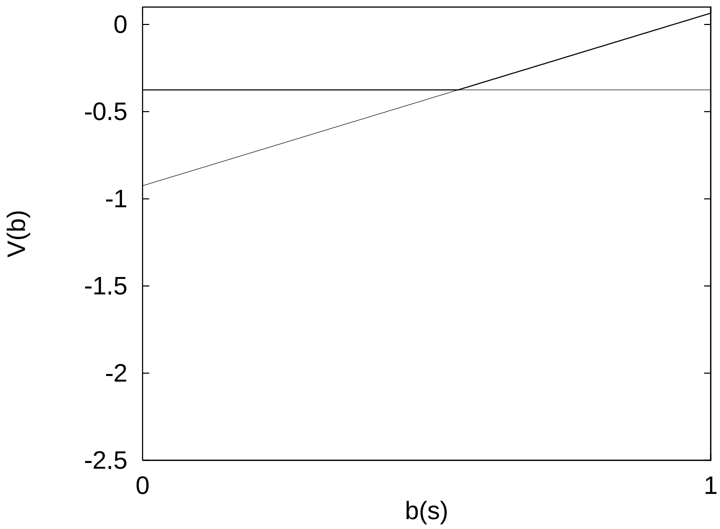


Figure F.1: Value function $V_1(\cdot)$.

2 **Steps-to-go:** First, considering action 0 we build the $\Gamma_2^{0,z}$ value functions from Γ_1 which yields

$$\Gamma_2^{0,0} = \left\{ \begin{array}{c} \left[\begin{array}{rr} 0.02262 & -1.03369 \end{array} \right] \\ \left[\begin{array}{rr} -0.26319 & -0.69406 \end{array} \right] \end{array} \right\}$$

$$\Gamma_2^{0,1} = \left\{ \begin{array}{c} \left[\begin{array}{rr} 0.01008 & -0.77006 \end{array} \right] \\ \left[\begin{array}{rr} -0.02806 & -0.58719 \end{array} \right] \end{array} \right\} \quad .$$

Computing the full cross-sum, $\Gamma_2^{0,0} \oplus \Gamma_2^{0,1}$, we get 4 vectors, though one of these has an empty region as shown with the dashed line in Figure F.2. Thus we find that

$$\Gamma_2^{0} = \left\{ \begin{array}{c} \left[\begin{array}{rr} 0.03270 & -1.80375 \end{array} \right] \\ \left[\begin{array}{rr} -0.29125 & -1.28125 \end{array} \right] \\ \left[\begin{array}{rr} -0.00544 & -1.62088 \end{array} \right] \end{array} \right\} \quad .$$

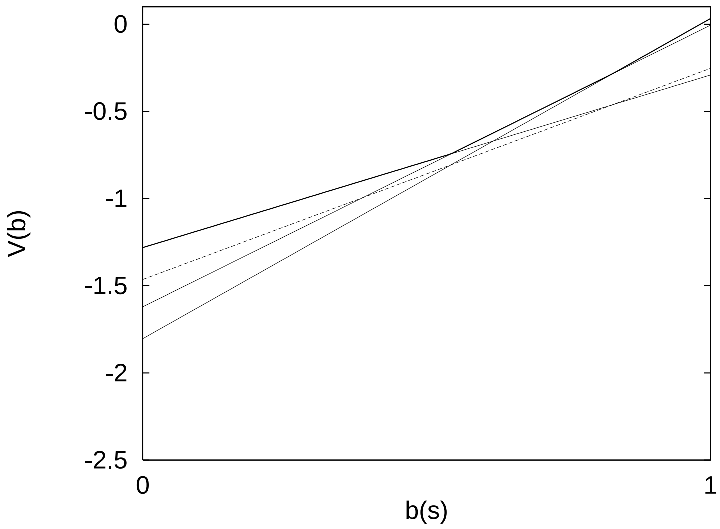


Figure F.2: Full cross-sum for $\Gamma_2^{0,0} \oplus \Gamma_2^{0,1}$ with one useless vector.

For action 1, the associated single action and observation sets are

$$\Gamma_2^{1,0} = \left\{ \begin{array}{c} [\ -0.44685 \quad -0.44685 \ ] \\ [\ -0.45469 \quad -0.45469 \ ] \end{array} \right\}$$

$$\Gamma_2^{1,1} = \left\{ \begin{array}{c} [\ -0.33665 \quad -0.33665 \ ] \\ [\ -0.27656 \quad -0.27656 \ ] \end{array} \right\} \ .$$

Since each of these vectors is a horizontal line, the full cross-sum contains only horizontal lines and the Γ_2^1 set will consist of a single vector which is the largest, i.e.,

$$\Gamma_2^1 = \{ \ [\ -0.72341 \quad -0.72341 \ ] \ \} \ .$$

Combining Γ_2^0 and Γ_2^1 we find that one of the vectors from Γ_2^0 has an empty region, making the final set

$$\Gamma_2 = \left\{ \begin{array}{c} [\ \ \ 0.03270 \quad -1.80375 \ ] \\ [\ -0.00544 \quad -1.62088 \ ] \\ [\ -0.72341 \quad -0.72341 \ ] \end{array} \right\} \ ,$$

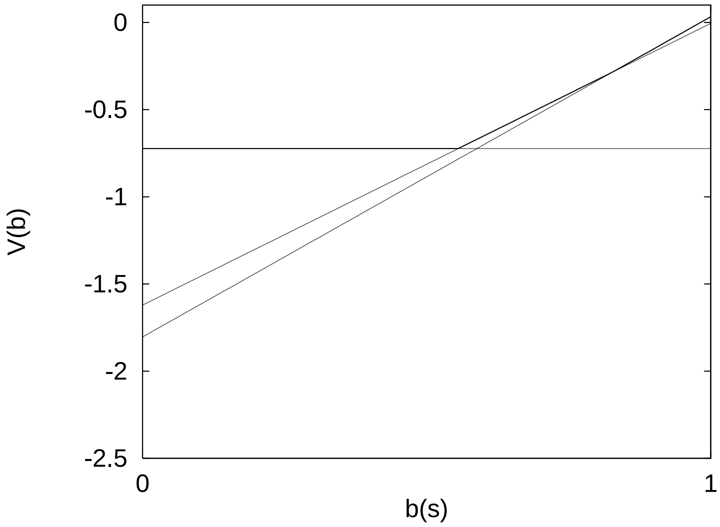


Figure F.3: Value function $V_2(\cdot)$.

which is shown in Figure F.3.

3 Steps-to-go: We start with

$$\Gamma_3^{0,0} = \left\{ \begin{array}{c} [\ -0.05512 \quad -1.57632\ ] \\ [\ -0.07154 \quad -1.46339\ ] \\ [\ -0.53791 \quad -0.90921\ ] \end{array} \right\}$$

$$\Gamma_3^{0,1} = \left\{ \begin{array}{c} [\ -0.02328 \quad -1.06225\ ] \\ [\ -0.02209 \quad -1.00144\ ] \\ [\ -0.08433 \quad -0.70303\ ] \end{array} \right\} ,$$

and generate the full cross-sum $\Gamma_3^{0,0} \oplus \Gamma_3^{0,1}$, which yields 9 vectors, shown in Figure F.4. Although it is difficult to see in the figure, all but four of these vectors have empty regions, resulting in

$$\Gamma_3^{0} = \left\{ \begin{array}{c} [\ -0.07721 \quad -2.57776\ ] \\ [\ -0.09364 \quad -2.46483\ ] \\ [\ -0.15587 \quad -2.16642\ ] \\ [\ -0.62224 \quad -1.61224\ ] \end{array} \right\} ,$$

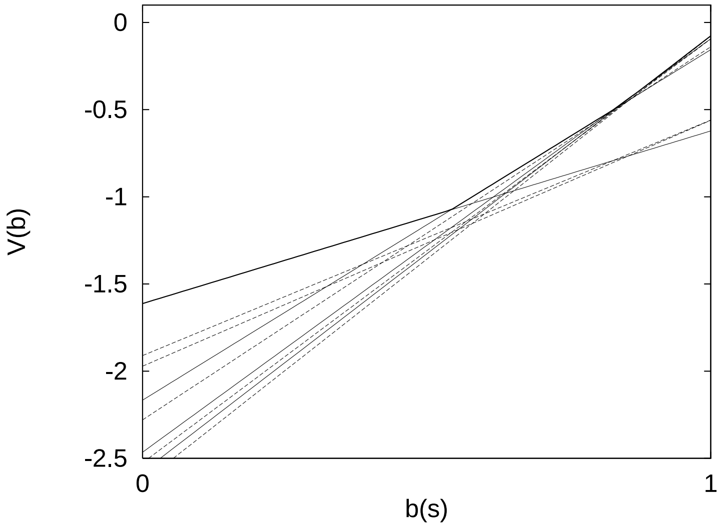


Figure F.4: Full cross-sum for $\Gamma_3^{0,0} \oplus \Gamma_3^{0,1}$ with 5 useless vector.

For action 1 we have

$$\Gamma_3^{1,0} = \left\{ \begin{array}{c} \left[\begin{array}{cc} -0.73121 & -0.73121 \end{array} \right] \\ \left[\begin{array}{cc} -0.69014 & -0.69014 \end{array} \right] \\ \left[\begin{array}{cc} -0.70293 & -0.70293 \end{array} \right] \end{array} \right\}$$

$$\Gamma_3^{1,1} = \left\{ \begin{array}{c} \left[\begin{array}{cc} -0.48504 & -0.48504 \end{array} \right] \\ \left[\begin{array}{cc} -0.45736 & -0.45736 \end{array} \right] \\ \left[\begin{array}{cc} -0.35931 & -0.35931 \end{array} \right] \end{array} \right\} .$$

As we had for the case of $a = 1$, $n = 2$, every vector in the two sets is a horizontal line, and the final set is simply the maximal line or

$$\Gamma_3^1 = \left\{ \left[\begin{array}{cc} -1.0495 & -1.0495 \end{array} \right] \right\} .$$

Combining Γ_3^0 and Γ_3^1 we find that one of the Γ_3^0 vectors has an empty

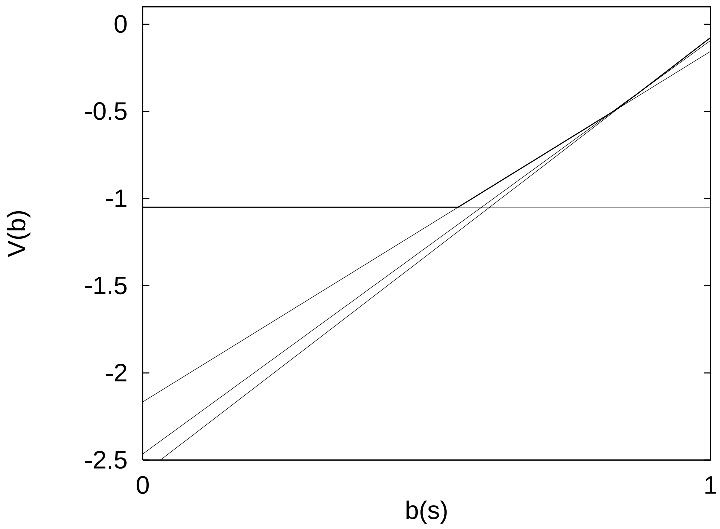


Figure F.5: Value function $V_3(\cdot)$.

region. This makes the final value function representation

$$\Gamma_3 = \left\{ \begin{array}{c} \left[\begin{array}{cc} -0.07721 & -2.57776 \end{array} \right] \\ \left[\begin{array}{cc} -0.09364 & -2.46483 \end{array} \right] \\ \left[\begin{array}{cc} -0.15587 & -2.16642 \end{array} \right] \\ \left[\begin{array}{cc} -1.04950 & -1.04950 \end{array} \right] \end{array} \right\},$$

which is shown in Figure F.5.

The basic incremental pruning algorithm continues in this same manner, though we terminate the example at this step. The next section shows the following DP step, but uses the witness algorithm.

F.2 Witness

We will discuss the operation of the witness algorithm for a single DP step. We will pick up where the incremental pruning algorithm left off, computing Γ_4 from Γ_3. An important point to be aware of is that we have rounded off the vectors to 5 decimal places, though the computations discussed used the full precision. Therefore, there may be appear to be slight discrepancies between the witness points and the vectors.

Action 0: The witness algorithm begins by selecting any information state, b, generating its vector, $\gamma_n^a(b)$, and adding that vector's neighbors to Υ. It will be useful to first show the $\Gamma_4^{0,z}$ sets, since we ill need them in the discussion:

$$\Gamma_4^{0,0} = \left\{ \begin{array}{ll} [\ -0.18279 \quad -2.05426\ ] & (\gamma_{4,0}^{0,0}) \\ [\ -0.18775 \quad -1.98453\ ] & (\gamma_{4,1}^{0,0}) \\ [\ -0.21455 \quad -1.80026\ ] & (\gamma_{4,2}^{0,0}) \\ [\ -0.79503 \quad -1.11057\ ] & (\gamma_{4,3}^{0,0}) \end{array} \right\}$$

$$\Gamma_4^{0,1} = \left\{ \begin{array}{ll} [\ -0.06311 \quad -1.31960\ ] & (\gamma_{4,0}^{0,1}) \\ [\ -0.06146 \quad -1.28206\ ] & (\gamma_{4,1}^{0,1}) \\ [\ -0.05952 \quad -1.18284\ ] & (\gamma_{4,2}^{0,1}) \\ [\ -0.13699 \quad -0.81146\ ] & (\gamma_{4,3}^{0,1}) \end{array} \right\} .$$

Notice that we have labeled each vector so that each vector constructed can be referenced to the vectors that were used to construct it, where $\gamma_{n,i}^{a,z}$ is the vector for action a, observation z, DP step n and is the i^{th} vector in the set $\Gamma_n^{a,z}$. We arbitrarily choose $b = [\ 0.0\ 1.0\ ]$ to start and find

$$\begin{aligned} \gamma_4^0([\ 0.0\ 1.0\ ]) &= \gamma_{4,3}^{0,0} + \gamma_{4,3}^{0,1} \\ &= [\ -0.93203\ -1.92203\ ], \end{aligned}$$

which is added to $\widehat{\Gamma}$. With $|\Gamma_4^{0,0}| = 4$ and $|\mathcal{Z}| = 2$, this vector has $|\mathcal{Z}|(|\Gamma_n^{a,z}| -$

1) = 6 neighbors which are added to Υ

$$\Upsilon = \mathcal{N}(\gamma_4^0([\,0.0\;1.0\,]))$$
$$= \left\{ \begin{array}{ll} [\;-0.31978 \quad -2.86572\;] & (\gamma_{4,0}^{0,0} + \gamma_{4,3}^{0,1}) \\ [\;-0.32475 \quad -2.79599\;] & (\gamma_{4,1}^{0,0} + \gamma_{4,3}^{0,1}) \\ [\;-0.35155 \quad -2.61172\;] & (\gamma_{4,2}^{0,0} + \gamma_{4,3}^{0,1}) \\ [\;-0.85814 \quad -2.43017\;] & (\gamma_{4,3}^{0,0} + \gamma_{4,0}^{0,1}) \\ [\;-0.85650 \quad -2.39262\;] & (\gamma_{4,3}^{0,0} + \gamma_{4,1}^{0,1}) \\ [\;-0.85455 \quad -2.29340\;] & (\gamma_{4,3}^{0,0} + \gamma_{4,2}^{0,1}) \end{array} \right\} .$$

We now enter the loop and select an item from Υ. Arbitrarily, we will always select the first item as listed and add items to the end of the list. Selecting $v = [\,-0.31978\; -2.86572\,]$. First, we check $R(v, \widehat{\Gamma})$ and, as shown in Figure F.6 with a dashed line, we find that it is not empty. Since the `findRegionPoint` LP maximizes the difference between v and $\widehat{\Gamma}$, it returns the witness point $b = [\,1.0\; 0.0\,]$. With this witness point we find its maximal vector to be

$$\gamma_4^0([\,1.0\;0.0\,]) = \gamma_{4,0}^{0,0} + \gamma_{4,2}^{0,1}$$
$$= [\,-0.24231\; -3.23710\,]\ ,$$

with neighbors

$$\mathcal{N}(\gamma_4^0(b)) = \left\{ \begin{array}{ll} [\;-0.24590 \quad -3.37387\;] & (\gamma_{4,0}^{0,0} + \gamma_{4,0}^{0,1}) \\ [\;-0.24425 \quad -3.33632\;] & (\gamma_{4,0}^{0,0} + \gamma_{4,1}^{0,1}) \\ [\;-0.31978 \quad -2.86572\;] & (\gamma_{4,0}^{0,0} + \gamma_{4,3}^{0,1}) \\ [\;-0.24728 \quad -3.16737\;] & (\gamma_{4,1}^{0,0} + \gamma_{4,2}^{0,1}) \\ [\;-0.27408 \quad -2.98310\;] & (\gamma_{4,2}^{0,0} + \gamma_{4,2}^{0,1}) \\ [\;-0.85455 \quad -2.29340\;] & (\gamma_{4,3}^{0,0} + \gamma_{4,2}^{0,1}) \end{array} \right\} .$$

Adding those neighbors to Υ that are not already in Υ and v back into

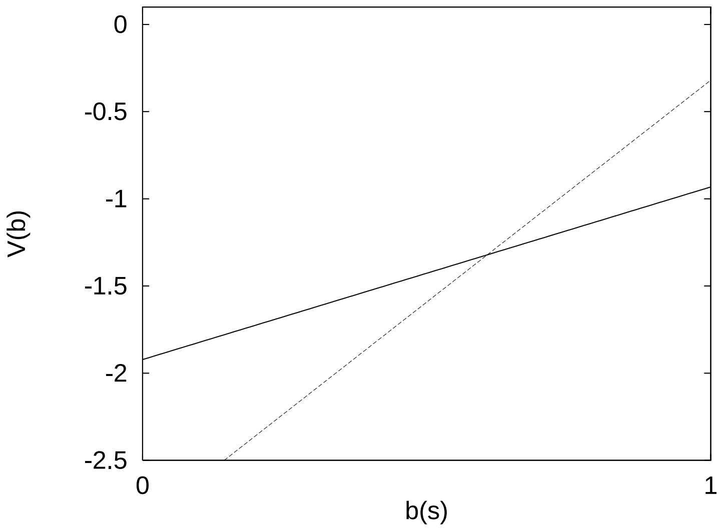


Figure F.6: An agenda item with a non-empty region over $\widehat{\Gamma}$.

Υ we have

$$\widehat{\Gamma} = \left\{ \begin{array}{c} [\ -0.93203 \quad -1.92203\ ] \\ [\ -0.24231 \quad -3.23710\ ] \end{array} \right\}$$

$$\Upsilon = \left\{ \begin{array}{c} [\ -0.32475 \quad -2.79599\ ] \\ [\ -0.35155 \quad -2.61172\ ] \\ [\ -0.85814 \quad -2.43017\ ] \\ [\ -0.85650 \quad -2.39262\ ] \\ [\ -0.85455 \quad -2.29340\ ] \\ [\ -0.24590 \quad -3.37387\ ] \\ [\ -0.24425 \quad -3.33632\ ] \\ [\ -0.24728 \quad -3.16737\ ] \\ [\ -0.27408 \quad -2.98310\ ] \\ [\ -0.31978 \quad -2.86572\ ] \end{array} \right\}.$$

Returning to the top of the loop, we select $v = [\ -0.32475\ \ -2.79599\ ]$ and compare it to $\widehat{\Gamma}$ which yields a non-empty region as shown in Figure F.7. Again, because `findRegionPoint` tries to find the point of maximal difference, it will return the point where the two vectors in $\widehat{\Gamma}$ intersect, i.e.,

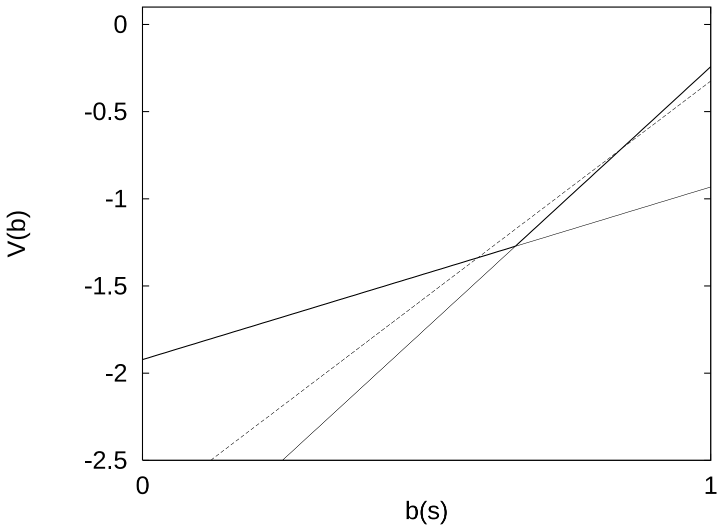


Figure F.7: Another agenda item with a non-empty region over $\widehat{\Gamma}$.

$b = [\ 0.65597\ 0.34403\ ]$. We get

$$\begin{aligned}\gamma_4^0(b) &= \gamma_{4,2}^{0,0} + \gamma_{4,3}^{0,1} \\ &= [\ -0.35155\ \ -2.61172\ ]\end{aligned}$$

with neighbors

$$\mathcal{N}(\gamma_4^0(b)) = \left\{ \begin{array}{ll} [\ -0.31978 \quad -2.86572\ ] & (\gamma_{4,0}^{0,0} + \gamma_{4,3}^{0,1}) \\ [\ -0.32475 \quad -2.79599\ ] & (\gamma_{4,1}^{0,0} + \gamma_{4,3}^{0,1}) \\ [\ -0.27767 \quad -3.11987\ ] & (\gamma_{4,2}^{0,0} + \gamma_{4,0}^{0,1}) \\ [\ -0.27602 \quad -3.08232\ ] & (\gamma_{4,2}^{0,0} + \gamma_{4,1}^{0,1}) \\ [\ -0.27408 \quad -2.98310\ ] & (\gamma_{4,2}^{0,0} + \gamma_{4,2}^{0,1}) \\ [\ -0.93203 \quad -1.92203\ ] & (\gamma_{4,3}^{0,0} + \gamma_{4,3}^{0,1}) \end{array} \right\} .$$

Adding these neighbors and υ into Υ we get

$$\widehat{\Gamma} = \left\{ \begin{array}{c} \left[\begin{array}{cc} -0.93203 & -1.92203 \end{array} \right] \\ \left[\begin{array}{cc} -0.24231 & -3.23710 \end{array} \right] \\ \left[\begin{array}{cc} -0.35155 & -2.61172 \end{array} \right] \end{array} \right\}$$

$$\Upsilon = \left\{ \begin{array}{c} \left[\begin{array}{cc} -0.35155 & -2.61172 \end{array} \right] \\ \left[\begin{array}{cc} -0.85814 & -2.43017 \end{array} \right] \\ \left[\begin{array}{cc} -0.85650 & -2.39262 \end{array} \right] \\ \left[\begin{array}{cc} -0.85455 & -2.29340 \end{array} \right] \\ \left[\begin{array}{cc} -0.24590 & -3.37387 \end{array} \right] \\ \left[\begin{array}{cc} -0.24425 & -3.33632 \end{array} \right] \\ \left[\begin{array}{cc} -0.24728 & -3.16737 \end{array} \right] \\ \left[\begin{array}{cc} -0.27408 & -2.98310 \end{array} \right] \\ \left[\begin{array}{cc} -0.31978 & -2.86572 \end{array} \right] \\ \left[\begin{array}{cc} -0.27767 & -3.11987 \end{array} \right] \\ \left[\begin{array}{cc} -0.27602 & -3.08232 \end{array} \right] \\ \left[\begin{array}{cc} -0.93203 & -1.92203 \end{array} \right] \\ \left[\begin{array}{cc} -0.32475 & -2.79599 \end{array} \right] \end{array} \right\} .$$

Returning to the top of the loop, the next vector we remove from Υ is $\upsilon = [\,-0.35155\ \ -2.61172\,]$, but we find that $\upsilon \in \widehat{\Gamma}$, so we discard this, return to the top of the loop and remove the next item to $\upsilon = [\,-0.85814\ \ -2.43017\,]$. As shown in Figure F.8, the $R(\upsilon, \widehat{\Gamma}) = \emptyset$ and we return to selecting items from Υ. We find that the next 4 items selected, assuming we select them in order they are listed above, all have empty regions over $\widehat{\Gamma}$. In all, 6 vectors have been removed from Υ without adding anything to $\widehat{\Gamma}$. The vectors removed are

$$\begin{array}{c} \left[\begin{array}{cc} -0.35155 & -2.61172 \end{array} \right] \\ \left[\begin{array}{cc} -0.85814 & -2.43017 \end{array} \right] \\ \left[\begin{array}{cc} -0.85650 & -2.39262 \end{array} \right] \\ \left[\begin{array}{cc} -0.85455 & -2.29340 \end{array} \right] \\ \left[\begin{array}{cc} -0.24590 & -3.37387 \end{array} \right] \\ \left[\begin{array}{cc} -0.24425 & -3.33632 \end{array} \right] \end{array}$$

Not until $\upsilon = [\,-0.24728\ \ -3.16737\,]$ do we find a non-empty region. In this case, although the region is quite small, `findRegionPoint` returns

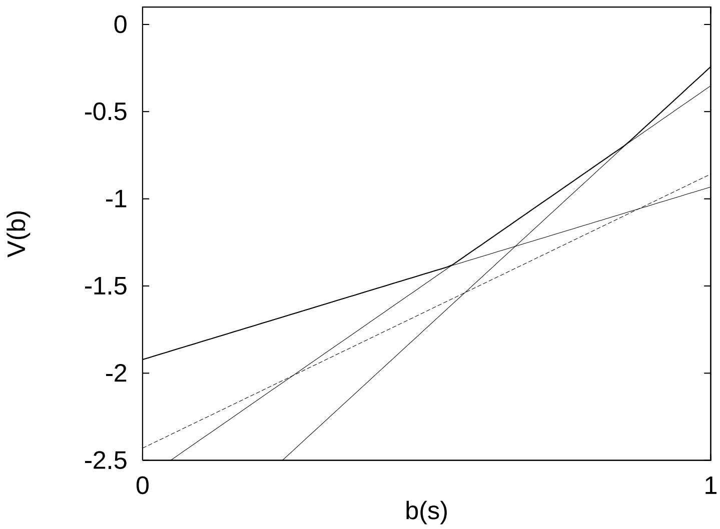


Figure F.8: An agenda item with an empty region over $\widehat{\Gamma}$.

$b = [\ 0.85131\ 0.14869\ ]$ and we generate

$$\begin{aligned}\gamma_4^0(b) &= \gamma_{4,1}^{0,0} + \gamma_{4,2}^{0,1} \\ &= [\ -0.24728\ \ -3.16737\ ]\ \ ,\end{aligned}$$

which just happens to be the same vector as v.

At this point, following the algorithm to the letter would require adding this vector both to $\widehat{\Gamma}$ and Υ. However, putting it in $\widehat{\Gamma}$ guarantees that we will not check this vector when we later remove it from Υ, since the algorithm specifically checks for this condition. Thus, we will elect not to add them to Υ.

Even if we do not add the vector v to Υ, we are still required to add its neighbors to Υ. We have

$$\mathcal{N}(\gamma_4^0(b)) = \left\{ \begin{array}{lll} [\ -0.24231 \quad -3.23710\ ] & (\gamma_{4,0}^{0,0} + \gamma_{4,2}^{0,1}) & \text{(a)} \\ [\ -0.25087 \quad -3.30414\ ] & (\gamma_{4,1}^{0,0} + \gamma_{4,0}^{0,1}) & \text{(a)} \\ [\ -0.24922 \quad -3.26659\ ] & (\gamma_{4,1}^{0,0} + \gamma_{4,1}^{0,1}) & \text{(a)} \\ [\ -0.32475 \quad -2.79599\ ] & (\gamma_{4,1}^{0,0} + \gamma_{4,3}^{0,1}) & \text{(b)} \\ [\ -0.27408 \quad -2.98310\ ] & (\gamma_{4,2}^{0,0} + \gamma_{4,2}^{0,1}) & \text{(b)} \\ [\ -0.85455 \quad -2.29340\ ] & (\gamma_{4,3}^{0,0} + \gamma_{4,2}^{0,1}) & \text{(c)} \end{array} \right\} ,$$

which allows us to relate some of the optimization ideas from Section 3.2.3. Looking at the neighbor set above, we see three types of vectors which we have labeled (a), (b) and (c) above. We find:

(a) There are vectors which are not, and have never been in Υ. In this case there are three of these, and we have no choice but to add it to Υ.

(b) There are two vectors which are currently in the agenda. By nature of taking the union of the neighbor set and Υ, these will not contribute to making Υ any larger.

(c) There is one vector that is not currently in Υ, however, it previously was in Υ until it was removed, due to it producing an empty region. The algorithm, as it appears in Table 3.7, will add this to Υ, but this is not necessary. If the region was empty before, having added more vectors to $\widehat{\Gamma}$ cannot make this region non-empty. Thus, an optimization that can be added keeps track of vectors removed from Υ. We will use this optimization here and not add this vector to Υ.

With the optimizations in place, we end up adding only three vectors to

Υ giving

$$\widehat{\Gamma} = \left\{ \begin{array}{c} [\ -0.93203\ \ -1.92203\ ] \\ [\ -0.24231\ \ -3.23710\ ] \\ [\ -0.35155\ \ -2.61172\ ] \\ [\ -0.24728\ \ -3.16737\ ] \end{array} \right\}$$

$$\Upsilon = \left\{ \begin{array}{c} [\ -0.27408\ \ -2.98310\ ] \\ [\ -0.31978\ \ -2.86572\ ] \\ [\ -0.27767\ \ -3.11987\ ] \\ [\ -0.27602\ \ -3.08232\ ] \\ [\ -0.93203\ \ -1.92203\ ] \\ [\ -0.32475\ \ -2.79599\ ] \\ [\ -0.24231\ \ -3.23710\ ] \\ [\ -0.25087\ \ -3.30414\ ] \\ [\ -0.24922\ \ -3.26659\ ] \end{array} \right\}.$$

The next item selected at the top of the loop is $v = [\ -0.27408\ -2.98310\ ]$ which has a small, non-empty region, yielding $b = [\ 0.84200\ 0.15800\ ]$. We find that this item happens to be the maximal vector for this point and has

$$\mathcal{N}(\gamma_4^0(b)) = \left\{ \begin{array}{lll} [\ -0.24231\ \ -3.23710\ ] & (\gamma_{4,0}^{0,0} + \gamma_{4,2}^{0,1}) & \text{(b)} \\ [\ -0.24728\ \ -3.16737\ ] & (\gamma_{4,1}^{0,0} + \gamma_{4,2}^{0,1}) & \text{(c)} \\ [\ -0.27767\ \ -3.11987\ ] & (\gamma_{4,2}^{0,0} + \gamma_{4,0}^{0,1}) & \text{(b)} \\ [\ -0.27602\ \ -3.08232\ ] & (\gamma_{4,2}^{0,0} + \gamma_{4,1}^{0,1}) & \text{(b)} \\ [\ -0.35155\ \ -2.61172\ ] & (\gamma_{4,2}^{0,0} + \gamma_{4,3}^{0,1}) & \text{(c)} \\ [\ -0.85455\ \ -2.29340\ ] & (\gamma_{4,3}^{0,0} + \gamma_{4,2}^{0,1}) & \text{(c)} \end{array} \right\},$$

where we have again indicated the types. For this vector, since all neighbors are either in Υ or were previously in Υ, we do not have to add anything to the agenda. We now have

$$\widehat{\Gamma} = \left\{ \begin{array}{c} [\ -0.93203 \quad -1.92203\ ] \\ [\ -0.24231 \quad -3.23710\ ] \\ [\ -0.35155 \quad -2.61172\ ] \\ [\ -0.24728 \quad -3.16737\ ] \\ [\ -0.27408 \quad -2.98310\ ] \end{array} \right\}$$

$$\Upsilon = \left\{ \begin{array}{c} [\ -0.31978 \quad -2.86572\ ] \\ [\ -0.27767 \quad -3.11987\ ] \\ [\ -0.27602 \quad -3.08232\ ] \\ [\ -0.93203 \quad -1.92203\ ] \\ [\ -0.32475 \quad -2.79599\ ] \\ [\ -0.24231 \quad -3.23710\ ] \\ [\ -0.25087 \quad -3.30414\ ] \\ [\ -0.24922 \quad -3.26659\ ] \end{array} \right\}.$$

We will now find, selecting one vector at a time, that all the remaining items in Υ yield empty regions over $\widehat{\Gamma}$. When Υ has been exhausted, we are left with $\Gamma_4^0 = \widehat{\Gamma}$ and the witness algorithm is complete. The final value function, $V_4^0(\cdot)$, represented by Γ_4^0 is shown in Figure F.9.

Action 1: The witness algorithm proceeds in the same for action 1, except the structure of the solution makes this somewhat simpler. Instead of showing the full witness algorithm for this case, we show how one of the optimizations discussed in Section 3.2.3 can save a large amount of work.

Recall from Section 3.2.3 that the witness algorithm can be modified to initialize $\widehat{\Gamma}$ with vectors generated from any number of points. For this case, assume we select our set of point to be the information space simplex corners. After checking all (two) simplex corners, we find that $\gamma_4^1([\ 0.0\ 1.0\ ] = \gamma_4^1([\ 1.0\ 0.0\ ] = [\ -1.3561\ \ -1.3561\ ]$. We had mentioned that checking all the simplex corners is guaranteed to yield at least two vectors if and only if $|\Gamma_4^1| > 1$. From this we can conclude that we have already

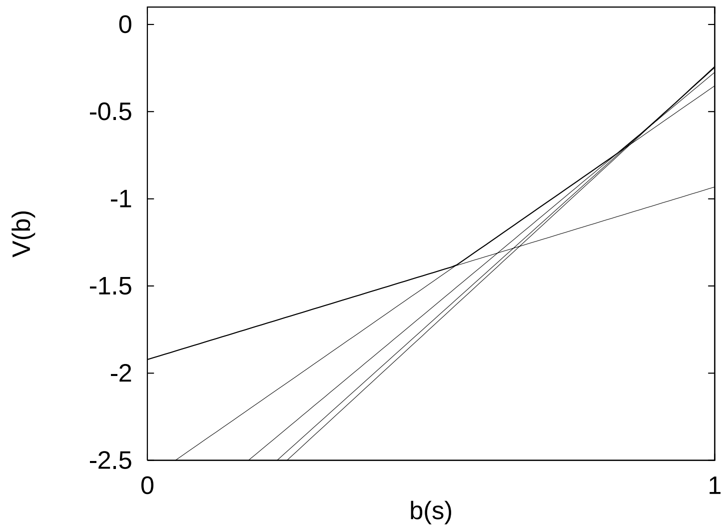


Figure F.9: The value function $V_4^0(\cdot)$ for the witness example.

determined Γ_4^1 and we never have to enter the witness loop.

Without this optimization, we would have been required to add all the neighbors of the vector to Υ and the witness loop would execute once for each neighbor (6 times), each time doing an LP only to find an empty region.

Merging Γ_4^0 and Γ_4^1 The final step for the $n = 4$ step is to merge the sets for all the actions. Figure F.10 shows that one vector from Γ_4^0 (with the dashed line) becomes useless.

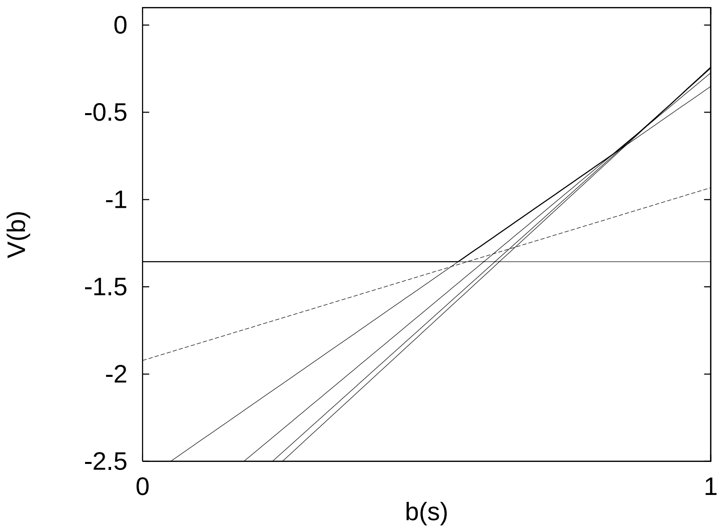


Figure F.10: The final value function $V_4(\cdot)$ for the witness example with one useless vector from Γ_4^0 shown.

Appendix G

Policy Graph Construction

In this section we will show the construction of a finite state controller for the infinite horizon version of the simple baseball example presented in Chapter 2 with Tables 2.1 and 2.6. This controller, which we call a *policy graph*, is derived from executing value iteration for a large horizon. We note that the technique shown in this section cannot always be applied, or may require a more complicated construction. This example is simply to illustrate how an optimal policy graph can be constructed for certain policies. Appendix D discusses the conditions on the policies required to be able to accomplish this, where the property required is are called finite transience .

There is a corresponding finite-horizon policy graph, which has a tree structure, where the infinite horizon policy graph can have cycles. We will use the general term policy graph for both in this appendix, allowing the context to disambiguate the two.

We begin with Figure G.1 which shows the finite horizon policy graph for the first 4 value iteration steps. Each node in this graph represents one

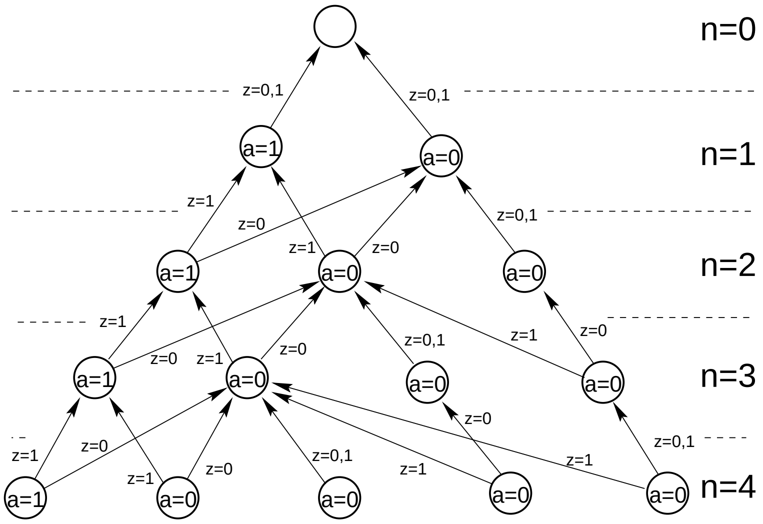


Figure G.1: Finite horizon policy graph structure.

of the vectors in Γ_n. The edges in the graph indicate which of the previous vectors where used in the construction of that particular vector; i.e., the choices made in the $\chi_{n-1}(\cdot)$ function of Equation 3.2. The labels on the nodes indicate the action associated with the vector.

Since each vector defines a region of information space, each node can also be viewed as representing a region. This graph also indicates the information state transformation function between these regions; if the information state is in the region of a vector and we perform its associated action, then the edges of the graph indicate which region the transformed information state will lie in for each possible observation. We have omitted the partition information to keep the figure uncluttered, but the nodes are ordered by the partitions so that the node corresponding to the vector that has $b = [\ 0.0\ 1.0\ ]$ in its region is on the far left. Note that the ordering is made possible by there being only two states, making the information state space essentially one-dimensional.

Aside from indicating which Γ_{n-1} vectors helped construct a vector or the information state transformation process, the main use of this graph is how it specifies the optimal finite horizon policy to follow. Given an information state and a certain number of steps to go, n, we can find the maximal vector (or simply see which region the information state is in), which will be one of the nodes in the graph from among those in horizon n. Following the action of the node label, we will get an observation and we follow the edge based on that observation to arrive at another node with one less step to go. In this way, the node indicates the action, and the edges indicate the next node and consequently, the next action. Therefore, given this finite horizon policy graph and the initial starting state, we can use it to optimally control the POMDP by selecting actions according to the current node, and moving in the graph according to the observations received.

Picking up where the algorithm examples left off in Appendix F, $n = 5$, we find that for all $n \geq 5$, the sizes of the Γ_n sets stop growing and for all $n \geq 4$ the parsimonious representation of $V_n(\cdot)$ is of size 5. By the 385^{th} step, the machine precision limitations no longer allows us to distinguish between $V_{384}(\cdot)$ and $V_{385}(\cdot)$. The infinite horizon value function, $V(\cdot)$ is shown in Figure G.2 and is represented by the set of vectors

$$\Gamma = \left\{ \begin{array}{c} [\ -5.7178259162 \quad -9.5605069575\ ] \\ [\ -7.0743380696 \quad -7.0743380696\ ] \\ [\ -5.8794976079 \quad -8.5678886466\ ] \\ [\ -5.7759198754 \quad -9.0644942135\ ] \\ [\ -5.7313101520 \quad -9.3711481510\ ] \end{array} \right\},$$

presented with more precision than the example in Appendix F.

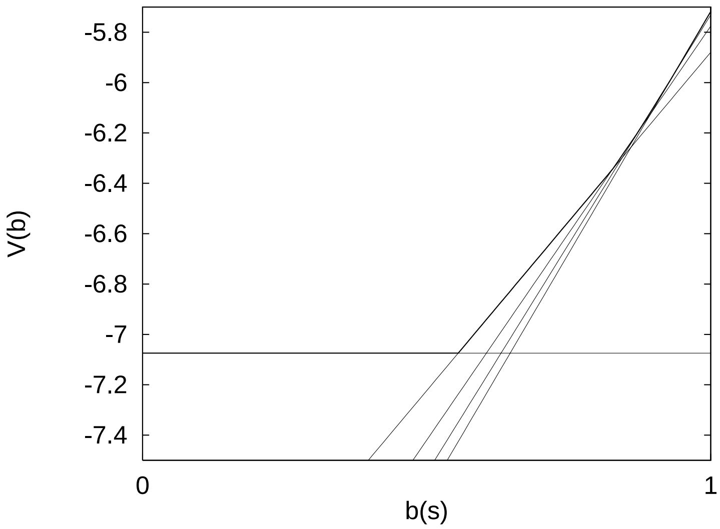


Figure G.2: Infinite horizon value function for baseball example.

Looking at the finite horizon policy graph, it has exactly the same structure for every step after the fourth as is shown in Figure G.3.

Figure G.4 shows the final partition imposed by the policy and the policy graph structure as it relates to these partitions. Note that the $n = 384$ and $n = 385$ portion of Figure G.3 is conveying exactly the same information as Figure G.4, which explicitly shows the policy graph structure on the information state space partitions.

At the infinite horizon, the vectors in one iteration are equivalent to the vectors in the next. With the repeated policy graph structure, we can redraw the edges from the nodes back onto themselves as is shown in Figure G.5. This can only be done because of the property of the optimal policy for this problem: this policy is finitely transient[1], as was discussed on Page 45.

[1] This policy is finitely transient with degree 4 [117].

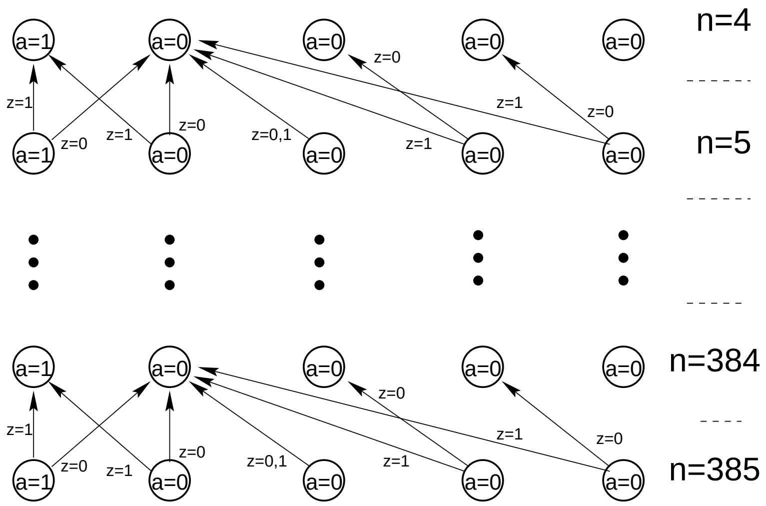


Figure G.3: Repeated policy graph structure.

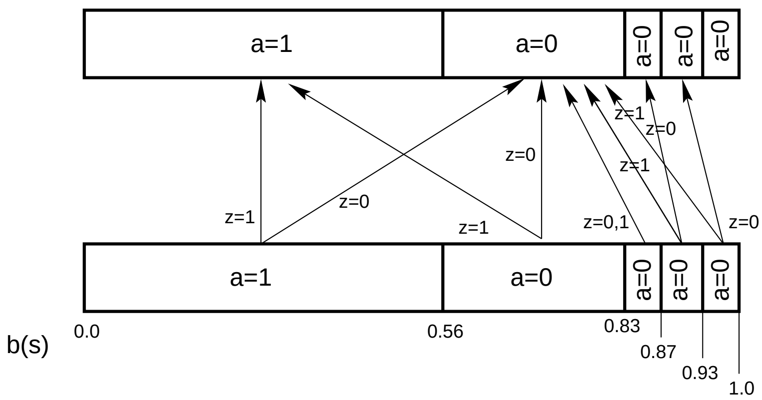


Figure G.4: Policy graph structure shown in relation to information state space partitions.

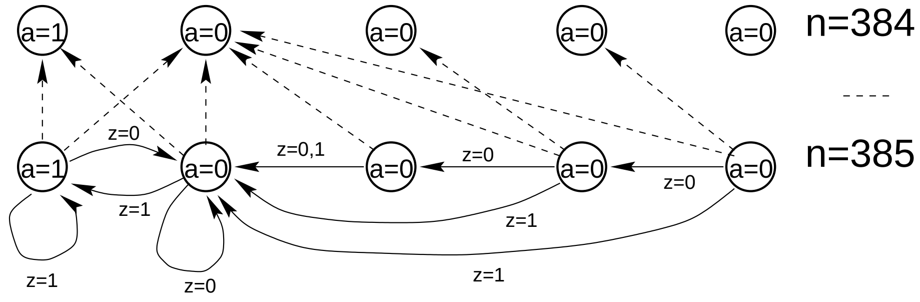


Figure G.5: Redrawing the edges for finitely transient policy.

The end result is an optimal controller that is a finite state machine as shown in Figure G.6. With this controller, we no longer need to track belief states, since the state machine transitions on the observation. In the figure, we have marked the starting node with an arrow, which represents the starting node for the initial belief state $b = [\ 0.9\ 0.1\ ]$, i.e., the manager initially believes that it is a good match-up for the pitcher with probability 0.9. Notice that the optimal policy to follow is not very realistic; this is simply a function of the simplicity of the problem which does not nearly capture the essence of the game of baseball. However, assuming the model is correct, using this small controller will yield optimal performance.

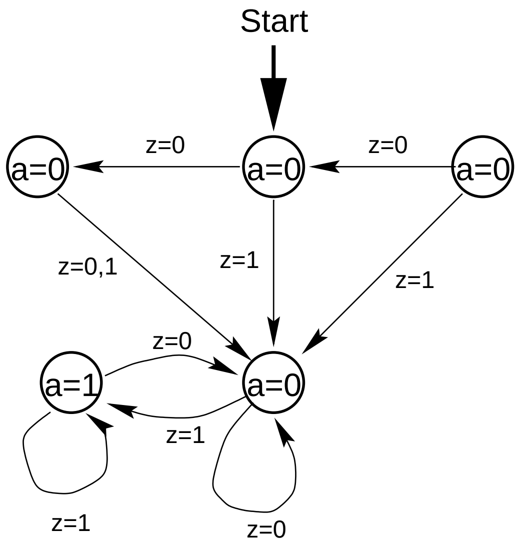


Figure G.6: Optimal infinite horizon controller for baseball example.

Appendix H

Example Domains

This appendix provides descriptions for some of the example domains used in the empirical comparisons[1].

[1] Every effort will be made to ensure that the actual problems used for the empirical results are available through the author or Brown University.

H.1 Large Baseball Domain

This domain is roughly based upon the example that appears in Howard's book [49] though others have applied decision theory to this domain [19]. It considers the strategic decisions of a baseball manager, for some number of innings, when their team is batting[2]. The discussion in here assumes some knowledge about the game of baseball. Appendix A provides enough discussion to understand the small baseball example of the main text, but it probably insufficient for full comprehension of this section. Note that the fundamental probabilities for this case are different than the simpler baseball example used in Chapter 2.

The choices available to the manager concern what the runners on base (if any) and the batter should do. The actions the manager has to choose from are

- `hit` - tells the batter to swing normally and the runners to wait and see the result before deciding to advance a base. This is typically what is chosen when there is no one on base or when the manager thinks the hitter is likely to get a hit.

- `bunt` - tells the batter to bunt the ball softly and the runners to start to advance as the pitch is being thrown. This decision is typically used when the manager is willing to sacrifice an out for advancing the runners a base. The drawback is that the batter is very likely to be out.

[2]As Howard does, we make no claim to the validity of the modeling assumptions, decisions or data toward the real situation in a baseball game.

- `hit-and-run` - tells the batter to swing normally, but for the runners to be moving as the pitch is delivered. The advantage here is that they will advance further on a hit and avoid a double-play. The disadvantage lies in the potential for a double-play on a strike-out, resulting from the runner being thrown out trying to steal.

- `steal-2nd` - tells the runner on first-base to attempt to steal second-base. This action is only valid if there is a runner on first and no runner on second.

- `steal-3rd` - tells the runner on second-base to attempt to steal third-base. This action is only valid if there is a runner on second and no runner on third. If there is also a runner on second-base, it has the effect of a double steal.

- `steal-home` - tells the runner on third to attempt to steal home. This is only valid if there is a runner on third, and if there are other runners, if has the effect of all of them attempting to steal.

Howard considers a completely observable situation, but here we add some hidden state to complicate the manager's decision. The hidden state consists of the quality of the pitcher, batter and catcher. All three of these players can either be in a good or bad state and their performance is directly related to their state. Thus there are 8 possible combination of the hidden state of these three players. For the specific parameters used, we have verified that these hidden state components would influence the optimal action selection if the state was fully observable.

There is actually only one observable component of the state space: the result of the last play. However, this component fully determines the values of a number of other components of the state space, since we have precisely defined the resulting situation for every initial situation and outcome. Thus these other components are, in effect, fully observable. The full set of effectively fully observable state components is

- The inning itself. The problem is extensible along this dimension. For the data shown, we only consider a single inning.
- The number of outs in the inning. Valid values are 0, 1 or 2. We do not need a 3 out state, since this is the same as the start of a new inning or the end of a game.
- The situation on the base-paths. Whether or not there is a runner on first-base, second-base or third-base. There are eight possible situations.
- How many players scored on the last play. This doesn't really have much effect on the current decision, but allows the rewards to be based upon the state, since a reward of 1 is received for each run that is scored. There are 5 valid values here, representing 0 through the maximum of four runs scoring on a single play.
- The result of the last play. This is the component that completely determines these other state variables and is directly related to the observations. The possible values here are discussed below.

In addition to all possible combinations of the above state variables, there is an extra state which is an absorbing state which represents the game being over. There are also the 8 combinations of hidden state discussed previously. Using only a single inning, this makes a total of 7681 states for the problem.

The observations are the results of the last play and there are 10 possible. When any of the three hitting actions are chosen, one of 8 observations are possible: `single`, `double`, `triple`, `home-run`, `base-on-balls`, `strike-out`, `fly-out`, `ground-out`, which all correspond to the usual baseball interpretation and whose semantics are described below. If one of the three stealing actions are chosen they there are only two possible observations: `stolen-base` and `caught-stealing` both of which pertain to the outcome of the lead runner[3].

To simplify both the description and coding of this example, we define outcome probabilities for a typical batter and then scale these probabilities accordingly to incorporate the states of the pitcher, batter and catcher.

Table H.1 shows the statistics that we used as a basis for this example. These translate into a probability of 0.336 for not making an out on a given play when the `hit` or `hit-and-run` action is chosen. This is the nominal value, and the actual value will be adjusted twice: once to account for the quality of the batter and once more to account for the quality of the pitcher.

We next define conditional probabilities for the type of non-out and the type of out as shown in Tables H.2 and H.3. Although the probability of making an out will be scaled according to the state of the batter and pitcher,

[3] Since the 8 hitting and 2 stealing observations are mutually exclusive, we only actually need 8 total observations in the model.

550 plate appearances			
non-outs		**outs**	
85	singles	55	strike-outs
25	doubles	155	fly-outs
5	triples	155	ground-outs
20	home-runs		
50	base on balls		

Table H.1: Statistics for a typical batter which are used as the basis for the probabilities in the baseball domain.

Outcome	$\Pr(\cdot \mid \texttt{hit})$
`single`	0.460
`double`	0.135
`triple`	0.027
`home-run`	0.108
`base-on-balls`	0.270

Table H.2: Conditional probabilities for "non-out" outcomes for the `hit` and `hit-and-run` action.

these conditional probabilities will not be affected.

As mentioned, we will scale the probability that the batter does not make an out by the quality of the pitcher and catcher. We first adjust the non-out probability to compensate for the batter. If the batter is good, then the probability is multiplied by 1.15 and if the batter is bad we multiply the non-out probability by 0.80. Note that this scaling requires the non-out probability to not be too close to 1. After the non-out probability has been

Outcome	$\Pr(\cdot \mid \texttt{hit})$
`strike-out`	0.16
`fly-out`	0.42
`ground-out`	0.42

Table H.3: Conditional probabilities for "out" outcomes for the `hit` and `hit-and-run` action.

Outcome	Pr(· | bunt)
`single`	1.0
`double`	0.0
`triple`	0.0
`home-run`	0.0
`base-on-balls`	0.0

Table H.4: Conditional probabilities for `hit` outcomes for the `bunt` action.

Outcome	Pr(· | bunt)
`strike-out`	0.05
`fly-out`	0.10
`ground-out`	0.85

Table H.5: Conditional probabilities for `out` outcomes for the `bunt` action.

adjusted for the batter, we adjust it for the quality of a pitcher with the factors 0.75 and 1.15 for a good and bad pitcher respectively.

For the `bunt` action we have a slightly different situation. Here we define the probability of making an out as 0.9 and assume that the quality of the pitcher and batter *do not* have an effect on this probability. For bunting, we use the condition probabilities shown in Tables H.4 and H.5.

Finally, for the actions corresponding to stealing a base, Table H.6 shows nominal success probabilities for stealing the various bases. These are unadjusted values and they are scaled by 0.8 if the catcher is good and 1.10 if the catcher is bad. For all steal actions except one, it is assumed that all runners attempt to steal with only the lead runner having the potential of being thrown out. The one exception is when there is a runner on first and a runner on third and the `steal-2nd` action is chosen. In this case, the runner on third does not advance regardless of the outcome of the stealing action.

Stealing	**Prob. of Success**
2nd	0.75
3rd	0.50
home	0.10

Table H.6: Stealing base probabilities prior to adjustment for the state of the catcher.

To complete the problem description, we must define the semantics of the outcomes of various actions given the initial situation. For each of the `single`, `double` and `triple` actions all base-runners advance the same number of bases as the batter, except if one of three conditions hold: either the action was `hit-and-run` or `bunt` or there are two outs. In these three cases, the runners advance one more base than the batter. Naturally, any that make it to home would result in the state where that many players scored. For a `home-run` all runners score and for a walk, only runners that are forced to advance will change position.

For a strike-out, no runners advance if the `hit` action is specified. However, if the action was either `hit-and-run` or `bunt`, if there are any runners on base and there are less than two outs, then the lead runner is also out, resulting in a double play.

For a `fly-out`, all runners stay put with the exception of a possible sacrifice fly. When there are less than two outs and a runner is on third-base, a `fly-out` will result in the player on third scoring with any other runners staying on their respective bases. The potential for a sacrifice fly does not apply to the `bunt` action. Here, the `fly-out` is interpreted as a short pop-up with no one advancing.

Finally, the `ground-out` outcome is very much dependent upon whether

Catcher	**Batter**	**Pitcher**	**Value**
bad	bad	bad	0.574512688
bad	bad	good	0.319788843
bad	good	bad	1.281795868
bad	good	good	0.512531047
good	bad	bad	0.558606169
good	bad	good	0.257343051
good	good	bad	1.275406063
good	good	good	0.489496861

Table H.7: Optimal completely observable values for one inning variation of the large baseball domain.

a `hit`, `bunt` or `hit-and-run` action is specified and where the runners are at the time. For a `bunt` and `hit-and-run`, we assume that the base-runners have gotten a jump and will advance a base on the out, which also precludes any double-play when the outcome is a ground-out. For a `hit` action, the ground-out has the potential to produce a double play. Whenever there is a runner on first and the `hit` action results in a ground-out, the runner on first is out at second-base and the batter is out at first-base. On a double play any runners on second or third advance a base, if the inning isn't over. If the base situation has no one on first, then the ground-out results in those runners remaining at their respective bases.

The discount factor used is 0.999 and the optimal completely observed values (the expected number of runs for a one inning game) for the 8 possible starting state are given in Table H.7. Note that the only unknown in the starting state pertains to the values of the hidden state variables.

H.2 Slotted Aloha

In a packet switched network, efficiency is gained by allowing multiple transmitters to share a common channel. However, the physical limitations of a channel allow only one packet to be transmitted at a time. If two or more transmitters attempt to send a packet at the same instant, the messages get garbled and a *collision* results. In this case, both packets are assumed to be backlogged and must be re-sent at a later time.

We make the simplifying assumptions that time is divided into fixed intervals called *slots*, packets can only be transmitted at the beginning of a slot and that all packets require exactly one time slot to be transmitted. All transmitting stations are synchronized with respect to the clock, but there is no other way for them to communicate.

The *slotted Aloha* protocol is a strategy for scheduling packet transmissions, where each packet waiting to be transmitted is transmitted with probability a [10]. If at a given time, there are s_b backlogged packets and all transmitting stations have access to the this number, then the optimal strategy is to transmit each backlogged packet with probability $1/s_b$ [4]. However, the transmission stations do not have access to the total backlog. The only thing the transmission stations have access to is the status of the channel and there are only three possible states of the channel: `idle` - no packets transmitted; `transmit` - a packet was successfully transmitted; `collision` - two or more packets collided.

[4]The probability of a successful transmission is $s_b a(1-a)^{s_b-1}$. Taking the derivative with respect to a and finding the critical points, we see that the successful transmission probability is maximized when $a = 1/s_b$.

For our example domain, we assume that packets arrive in the system according to a truncated Poisson distribution with mean 0.9. We fix the maximum number of arrivals at any slot to be 10 and the remaining probability mass of the Poisson distribution is given to the probability that no packet arrives. We also set a maximum number of backlogged messages in the system. This maximum number of backlogged messages is adjustable and we used the values 10 and 30 for the empirical comparisons.

We now discuss modeling this with a POMDP. The state of the system consists of the number of backlogged messages and the status of the channel for the previous transmission slot. The observations are the three possible states of the channel, `idle`, `transmit`, `collision`.

The actions are transmission probabilities which we must discretize to ensure a finite set. Since we know that choosing $a = 1/s_b$ maximizes the probability of a successful transmission, we choose the action set to be

$$\mathcal{A} = \left\{ \frac{1}{s_b} \mid s_b = 1, 2, \ldots, M \right\} \quad ,$$

where M is the maximum backlogged allowed.

The state transition probabilities account for the individual packet transmission probabilities for the given action and the Poisson arrival probability of new packets entering the system. When a packet is successfully transmitted, the backlog is reduced by one packet (and incremented by however many new packets arrived). When the channel is idle or there is a collision, then the backlog stays the same with the possible addition of new arrivals. We assume the observations of the channel status are deterministic based upon the last channel state.

For the rewards, we use a reward of zero when the maximum backlog is reached and $+1$ for each packet below maximum the system is at. Thus, the maximal reward is when there are no backlogged messages.

H.3 Machine Maintenance

We assume there is a machine with c internal components that is used to produce a part. The quality of the part produced is directly related to the condition or state of these internal components, which are only observable if the machine is disassembled. Each components can be in one of four conditions: `good` - the component is in good condition; `fair` - the component has some amount of wear, but would benefit from some maintenance; `bad` - the part is very worn and could use repairs; `broken` - the part is broken and must be replaced. Thus, the state of the machine is the combined state of the individual components and there are a total of 4^c states. For our examples we used $c = 4$.

The actions available to the decision maker are: `manufacture` - use the machine to produce parts for the day; `inspect` - allocate part of the day to disassembling the machine and inspecting the internal components; `repair` - allocate the day to maintenance of the machine's internal components; `replace` - replace the machine.

The state transitions depend upon the action. For the `manufacture` action, each component deteriorates with the probability 0.03 after each day of producing parts. Thus, a component in `good` condition may transition into the `fair` state, a component in `fair` condition may become `bad` and a `bad` component could become `broken`. A `broken` component does not deteriorate any further.

For the `repair` action we assume that the condition of each component improves with probability 0.8. A component in `broken` condition *cannot* be repaired, so it does not improve. A component in `bad` condition is likely to become `fair`, a component in `fair` condition is likely to become `good` with a `good` component not getting any better.

For the `inspect` action, the machine does not change state and for the `replace` action, all components deterministically move to the `good` condition.

The observations also depend upon the action taken. For the `manufacture` action, the observation is that either the machine produced good parts or bad parts for the day. In order for the machine to produce good parts, all components must perform properly during the day. A `good` component always performs properly during the day. A `fair` component produces properly with probability 0.95 and a `bad` component has probability 0.75 of performing properly. A `broken` component never performs properly.

For the `inspect` action, the observation is the composite of individual observations for each component, each of which is observed to be in a either a `good` or `bad` condition. The probabilities of the individual observations are dependent upon the actual condition of the component. A `good` component will yield a `good` observation with probability 0.97, a `fair` component looks `good` with probability 0.80, `bad` with probability 0.05 and broken is 0.02.

For the `repair` and `replace` actions, we assume that no observation is made, which is the same as deterministically getting the same observation, regardless of the state.

The rewards for the problem are 1 when good parts are manufactured for the day. Inspecting gives a -1 reward corresponding to the price to dismantle the machine and keep it idle. Repairing requires more effort and has a reward of -3. Finally, replacing the machine is the most costly and has a reward of -15.

H.4 Aircraft Identification (IFF)

This example is loosely based upon a model used by D'ambrosio and Fung [33]. The scenario involves an incoming aircraft where using various forms of sensors available at a base, the task is to determine if the aircraft is a threat or not. If the aircraft is a threat and nothing is done, then when the aircraft gets close enough, the base may be destroyed. However, if the aircraft is attacked and is not a threat (i.e., it is a `friendly` aircraft), then a significant penalty is accrued. The tension of deciding between the various sensors is that the better sensors tend to make the location of the base more easily identifiable or visible to the aircraft, while the more stealthy sensors tend to be less accurate. The sensors give information about both the aircraft's type and distance, though the distance information is generally more reliable than the aircraft type information.

State Space The state space of this problem is comprised of three main components:

- **aircraft type** - either the aircraft is a `friend` or it is a `foe`;
- **distance** - how far the aircraft is currently from the base discretized into an adjustable number, D, of distinct distances;
- **visibility** - a measure of how visible the base is to the approaching aircraft, which is discretized into 5 levels.

For the example domain we used $D = 10$. In addition to all combinations of these values, there are 4 extra states, which serve as zero-cost absorbing states:

- `base-safe` - results from a `friend` type aircraft reaching the base or an enemy reaching the base, but failing to destroy the base;
- `base-destroyed` - corresponds to a `foe` getting close enough and successfully attacking the base;
- `foe-destroyed` - results from successfully attacking a `foe` aircraft;
- `friend-destroyed` - results from attacking a `friend` aircraft and destroying it.

This brings the total number of states to

$$|\mathcal{S}| = 10D + 4 \quad .$$

State Transitions The transitions between the states depend upon the actions taken. There are $|\mathcal{A}| = 4$ actions available

- `active` - a sensing action using the more reliable sensor, which also renders the base more visible;
- `passive` - a sensing action using the less reliable sensor, but which does not make the base too visible to the incoming aircraft;
- `no-op` - employ no sensors;
- `attack` - attack the incoming aircraft;

The distance of the aircraft is measured in discrete locations from the base. Unless an absorbing state is reached, as described below, on a single

action	$\Pr(\mathbf{s^v_{t+1} = j \mid s^v_t = i})$		
	$\mathbf{j = i - 1}$	$\mathbf{j = i}$	$\mathbf{j = i + 1}$
`no-op`	0.25	0.75	0.00
`passive`	0.00	0.90	0.10
`active`	0.00	0.05	0.95
`attack`	0.00	0.20	0.80

Table H.8: Transition probabilities for the change in visibility level portion of the state.

step the aircraft will always advance a single discrete location with probability 0.8 and not advance with probability 0.2. It is impossible for the distance to get larger or to decrease by more than 1 discrete location. The change in distance is independent of the action chosen (assuming an absorbing state is not entered), the visibility level of the base and the type of aircraft.

The visibility level change depends upon the type of action chosen. There are 5 visibility levels and the visibility level can only change by at most one discrete unit per step. Letting s_t^v be the current visibility level and s_{t+1}^v be the next visibility level, Table H.8 shows the probability of the visibility level for each action. If the maximum/minimum visibility level is achieved, then the probability that the visibility level increases/decreases is zero. making a change in the visibility level of the base that much more probable. For the `attack` action, these probabilities are conditioned upon the aircraft not being destroyed and, for all of them, it is conditioned on the aircraft not destroying the base. Note that the aircraft type portion of the state never changes.

This defines the normal state transitions in terms of how the aircraft's distance changes and how the visibility level of the base changes. However, these transitions are all predicated on not arriving in one of the absorbing states.

Absorbing States For the `attack` action, the probability that the aircraft is destroyed is a function of how far away the aircraft is, independent of the base's visibility level or the aircraft's type. The probability that the aircraft is destroyed when the `attack` action is taken is given by

$$\frac{(D - s^d)^2}{D^2} \quad ,$$

where D is the number of discrete distances used in the model and s^d is the distance of the aircraft in terms of the number of discrete locations it is away from the base. The range on s^d is the interval $[\,0, D-1\,]$. If successful, the resulting state is either the `foe-destroyed` or `friend-destroyed` absorbing state depending on the aircraft type. If unsuccessful, then the state changes according to the previously discussed, though they are conditioned on the `attack` action failing, so their probabilities must be scaled by the probability that the aircraft was not destroyed.

If a `friend` aircraft is at $s^d = 0$, then on the next transition, with probability 1 the resulting state is the `base-safe` absorbing state. If a `foe` aircraft is at $s^d = 0$, then on the next transition, the state will be in the `base-destroyed` absorbing state with a probability proportional to the visibility level of the base given by

$$0.1 * s^v + 0.25 \quad ,$$

where s^v is the current visibility level of the base. This make the probability range for destroying the aircraft $[\,0.25, 0.65\,]$. If a `foe` aircraft fails to destroy the base, then the state becomes the `base-safe` absorbing state. Note this does not pertain to the `attack` action, since at $s^d = 0$, with probability 1 the plane is destroyed.

Observations The observations consist of two independent components: the aircraft type and the aircraft distance. In addition to these $2D$ possible observations, there are 2 additional observations: `nothing` which results from the `no-op` action with probability 1 and `absorb` which is the observation made in the four absorbing states with probability 1. We also assume that the `attack` action, if unsuccessful, returns the same information as the `active` sensing action, though the probabilities are then conditioned upon the attack failing and so must be multiplied by the probability of an unsuccessful attack.

For simplicity, we assume that the distance the sensors report is never more than 1 discrete location away from the true distance. An `active` sensing action will detect the true distance with probability 0.9 whereas a `passive` sensing action only detects the true distance with a 0.8 probability. The remaining probability mass for both actions are equally distributed among detecting the distance as being one location too close and one location too far. Since there are maximum and minimum distances, the boundary conditions are handled by adding the impossible distance's probability to the probability of detecting the true distance.

The sensors' detection of the plane type is independent of the distances reported by the sensors. An `active` sensing (or `attack`) action will detect the correct type with probability 0.8 and a `passive` sensing succeeds with

State	Reward
`base-safe`	0
`base-destroyed`	-100
`foe-destroyed`	+20
`friend-destroyed`	-30

Table H.9: Immediate rewards for entering the different absorbing states for the aircraft identification domain.

probability 0.6.

Immediate Rewards The only rewards that are defined are for transitions into one of the 4 absorbing states, corresponding to the final outcome. Table H.9 shows the immediate rewards for this domain.

H.5 Robot Navigation

The robot navigation problems concern themselves with a simplified robot with fairly crude sensors, navigating in an environment that is fairly structured as in an office environment. Although any reward structure can be incorporated into the POMDP, we use the simple idea of there being a single location the robot is trying to navigate to, which we refer to as the *goal* or *goal state*.

The crude sensors of the robot force it to have a simplified view of the world and in our case the robot only has the capability to make a simple observation directly in front of it and on either side of it. These three simple observations individually consist of either detecting a wall, a door, free space (`open`) or some undetermined sensor status. Thus, the full observation set for the robot consists of the four possible observations in the three directions, making a total of 64 possible observations. The observations the robot gets from these sensors are subject to noise and can result in the wrong observation being made with some probability.

The robot has a few fairly abstract actions which consist of moving forward, turning either left or right, doing nothing (`no-op`) and declaring that it has reached the goal state. Because of hardware limitations and other external conditions, the movements of the robot are subject to noise and are not completely reliable. Furthermore, the environment is assumed to consist of a finite number of discrete locations and a forward movement, if it succeeds, results in the robot moving one discrete location in the direction it is currently facing. The POMDP state of the robot consists of two components: its physical location in the discreteized world, and its orientation. We assume that the orientations themselves are discretized into four possible values, roughly corresponding to the four main compass directions. The actual number of states depends upon the actual physical layout of the environment. There is the addition of a zero-cost absorbing state which is entered when the robot issues the *declare-goal* action.

The immediate rewards for this POMDP are zero for all state-action pairs except for the pair consisting of declaring the goal when the robot is in the goal state. For this the immediate reward it receives is 1. If the robot declares itself to be in the goal when it is not, a penalty in the form of a -1 reward is received.

The POMDP models for a robot navigation problem are easy to specify compactly, despite the potential for a fairly large state space. The reason is that for the most part, the observation and transition probabilities are independent of the actual physical layout of the navigation domain. The only dependency on the transitions and observations is the local configuration immediately surrounding the robot's current location. Therefore, given the layout of the physical arrangement of the discretized environment and local transition and observation probabilities and semantics, the full transition and observation function of the POMDP is completely determined. This should become clearer below.

Transition Probabilities

To define the semantics of a robot's movement we will define some *primitive actions*, which should not be confused with the actions the robot executes,

Action	Outcome (probabilities)
move-forward	N (0.11), F (0.88), F-F (0.01)
turn-left	N (0.05), L (0.9), L-L (0.05)
turn-right	N (0.05), R (0.9), R-R (0.05)
no-op	N (1.0)
declare-goal	A (1.0)

Table H.10: Action probabilities for robot actions in terms of primitive actions.

i.e. those in the POMDP model. The primitive actions are:

- N - no robot movement,
- F - robot movement forward one discrete location,
- L - a change in robot orientation 90 degrees leftward and
- R - a change in robot orientation 90 degrees to the right.
- A - the robot, conceptually, goes into the absorbing state.

With these primitive actions we can specify the noise model for a given robot action by given a sequence of primitive actions and a probability that that sequence results. For example, consider Table H.10 where we see the `move-forward` action specified as having three possible outcomes: with probability 0.11 the robot will not change its state (location and orientation) at all; with probability 0.88 the `move-forward` action succeeds in moving the robot forward one location; and finally, with probability 0.1 the robot actually moves one location too far, which corresponds to two primitive F actions. The remaining actions are interpreted similarly.

The only complication that arises is that a particular location's configuration could render some of the possible outcomes impossible. The semantics we define is that the sequence of primitive actions proceeds are far as possible until an infeasible primitive action is encountered. The probability mass of the sequence of primitive actions is then added to the transition probability between the starting state and the final resulting state. For example, using the probabilities from Table H.10, suppose the robot was in a location where it could move forward one location, but a wall blocked it from moving forward two locations. The probability that the robot moves forward one location would be 0.89 which corresponds to the sum of two of the primitive action sequences, since the F-F sequence can only progress as far as one forward movement.

Observation Probabilities

There are only four basic things the robot can see: a wall, a door, open space or `undetermined`. We can completely specify the observational probabilities with a small table of conditional probabilities and some simple semantics for their application.

We consider every discrete location in the environment to either being part of hallway or part of a room. Between two adjacent locations of the same type, the space is open, navigable and the robot would, without noise, observe an open space in that direction. When two different types of locations are adjacent, we assume that it is navigable, but that the robot must pass through a door and so the observation received by the robot, again without noise, would be a door in that direction.

With these semantics, a small table of condition probabilities can completely specify the observational probabilities for the POMDP model. As an

Actual z_a	**Observed** z_o	$P(z_o \mid z_a)$
wall	wall	0.90
wall	open	0.05
wall	doorway	0.05
wall	undetermined	0.00
open	wall	0.03
open	open	0.90
open	doorway	0.07
open	undetermined	0.00
doorway	wall	0.15
doorway	open	0.15
doorway	doorway	0.70
doorway	undetermined	0.00
undetermined	undetermined	1.00

Table H.11: Conditional observation probabilities used to construct the observation probabilities.

example, Table H.11 gives the conditional probabilities for each observation based upon the true configuration and what its sensors are liable to report.

Thus, the full probability for a given observation in a given state can be computed by examining the locational layout to see what the true observations would be for that state, then computing the conditional probabilities once for the three directions its sensors report it, and finally multiplying these all together. Note that this assumes that the observations are all independent, which is not necessarily a valid assumption for a real robot.

Specific Domains

All of the example POMDPs constructed used all the previous rules for building the POMDP model. The only variables are the layout of the discretized locations, the initial state and the goal state. These are problem specific and described in Section 6.7.3.

Appendix I

Extra Data Tables

This appendix contains extra tables for the empirical studies done through this thesis. They are included here for completeness, and to keep the main body of text uncluttered.

I.1 Exact Algorithms

Section 4.9.1 showed most of the data in the form of a line graph, here we present the actual numbers and have done a simple two-sided T-test to gauge the significance of the differences. Tables I.1 through I.3 show the total number of LPs for constructing the Γ_n^a sets and Tables I.4 through I.6 show the total number of constraints.

Obs.	IpRr	IpNcs	TwoPass	Witness
3	150.800	117.160	525.520	564.840
4	351.240	298.800	1176.440	1255
5	807.400	733.800	2740	2899.400
6	1377.760	1282.520	4797.800	5053.440
7	2989.726	2880.493	10896.137	11368.137
8	3917.040	3820.560	13963.800	14548.640
9	5710.680	5585.680	21989.560	22524.760
10	8775.960	8674.360	29999.680	30482.960
11	13977.360	13565.480	52762.800	38906.960
12	20639.200	18584.040	77682.400	46657.440
13	25558.920	22982.640	98905.280	50200.840
14	32469.880	26716.560	1.513e+05	52681.600
15	44727.080	31763.800	2.002e+05	50458.840

Table I.1: Total LPs for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{S}| = 7$. T-test with $p = 0.95$.

States.	IpRr	IpNcs	TwoPass	Witness
3	537.160	$\boxed{508.120}$	1038.680	1100
4	$\boxed{1070.440}$	$\boxed{1014.760}$	2696.240	2829.280
5	$\boxed{1423.560}$	$\boxed{1359.040}$	4065.120	4265.440
6	$\boxed{1589.400}$	$\boxed{1508.840}$	4613	4848.600
7	$\boxed{2989.726}$	$\boxed{2880.493}$	10896.137	11368.137
8	$\boxed{3965.840}$	$\boxed{3840}$	16047.080	16088.520
9	$\boxed{4882.600}$	$\boxed{4730.720}$	20933.840	21764.960
10	$\boxed{5382.600}$	$\boxed{5233.880}$	24049.320	24175.160
11	$\boxed{7344.480}$	$\boxed{7156.400}$	36143.920	30195.200
12	$\boxed{10102.960}$	$\boxed{8761.760}$	56960.440	32591.280
13	$\boxed{10987.560}$	$\boxed{9905.840}$	60794.320	36203.160
14	$\boxed{14982.440}$	$\boxed{13116.880}$	82408.280	32967.600
15	16849.280	$\boxed{14395.840}$	1.056e+05	39311.640

Table I.2: Total LPs for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{Z}| = 7$. T-test with $p = 0.95$.

States/Obs.	IpRr	IpNcs	TwoPass	Witness
3	$\boxed{75.391}$	$\boxed{63.913}$	178.304	194.870
4	$\boxed{221}$	$\boxed{195.565}$	566.913	611.522
5	$\boxed{589.783}$	$\boxed{540.783}$	1726	1831.348
6	$\boxed{1103.391}$	$\boxed{1041.435}$	3460.087	3645.435
7	$\boxed{2989.726}$	$\boxed{2880.493}$	10896.137	11368.137
8	$\boxed{7098.652}$	$\boxed{6957.870}$	29132.261	29929.391
9	$\boxed{17539.261}$	$\boxed{15900}$	80523.304	43977.261
10	35760.391	$\boxed{22676.217}$	1.833e+05	39589.043
11	48863.652	$\boxed{24591.304}$	3.454e+05	33812.348
12	40611.565	$\boxed{22932.609}$	4.442e+05	29497.696
13	42541	$\boxed{24036.304}$	4.267e+05	$\boxed{27809.304}$
14	34272.304	$\boxed{18397.130}$	4.820e+05	$\boxed{19970.087}$
15	32184.696	$\boxed{16241.087}$	4.114e+05	$\boxed{17003.174}$

Table I.3: Total LPs for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{S}| = |\mathcal{Z}|$. T-test with $p = 0.95$.

Obs.	IpRr	IpNcs	TwoPass	Witness
3	974.840	1758.880	5481.840	13025.280
4	2975.520	7217.680	16206.160	53991.840
5	10570.640	29034.960	46633.680	2.255e+05
6	20453.400	62946.840	91369.920	5.175e+05
7	70838.438	2.769e+05	2.646e+05	2.395e+06
8	1.094e+05	4.373e+05	3.527e+05	3.461e+06
9	2.270e+05	9.157e+05	6.494e+05	8.684e+06
10	4.054e+05	1.538e+06	9.045e+05	1.199e+07
11	1.090e+06	3.889e+06	1.892e+06	1.937e+07
12	2.048e+06	7.073e+06	3.112e+06	2.874e+07
13	3.316e+06	1.0e+07	4.152e+06	3.326e+07
14	4.779e+06	1.190e+07	6.854e+06	3.619e+07
15	7.523e+06	1.507e+07	9.651e+06	3.845e+07

Table I.4: Total constraints for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{S}| = 7$. T-test with $p = 0.95$.

States.	IpRr	IpNcs	TwoPass	Witness
3	3812.280	7619.280	18035.360	30382.160
4	11495.120	33590.240	58241.040	1.793e+05
5	19550.640	57306.240	84982.800	3.702e+05
6	23230.640	68220.240	94734.880	4.509e+05
7	70838.438	2.769e+05	2.646e+05	2.395e+06
8	1.337e+05	5.717e+05	4.121e+05	5.263e+06
9	1.678e+05	6.841e+05	5.460e+05	6.626e+06
10	2.096e+05	9.145e+05	6.364e+05	8.722e+06
11	4.445e+05	2.107e+06	1.041e+06	1.4e+07
12	9.587e+05	3.138e+06	1.806e+06	1.690e+07
13	8.671e+05	3.879e+06	1.921e+06	1.951e+07
14	1.411e+06	6.367e+06	2.666e+06	2.030e+07
15	1.914e+06	6.514e+06	3.332e+06	2.527e+07

Table I.5: Total constraints for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{Z}| = 7$. T-test with $p = 0.95$.

States/Obs.	IpRr	IpNcs	TwoPass	Witness
3	$\boxed{368.478}$	$\boxed{474.522}$	1617.565	2090
4	$\boxed{1369.087}$	2408.783	6688.609	12121.304
5	$\boxed{5345.087}$	12772.087	28049.826	81469.826
6	$\boxed{13605.913}$	42373.261	65884.609	2.778e+05
7	$\boxed{70838.438}$	2.769e+05	2.646e+05	2.395e+06
8	$\boxed{3.573e+05}$	1.443e+06	8.391e+05	1.326e+07
9	$\boxed{1.615e+06}$	6.742e+06	2.817e+06	2.983e+07
10	$\boxed{5.692e+06}$	1.264e+07	$\boxed{7.390e+06}$	3.232e+07
11	$\boxed{1.042e+07}$	1.679e+07	1.518e+07	3.385e+07
12	$\boxed{1.056e+07}$	1.553e+07	2.077e+07	3.0e+07
13	$\boxed{1.051e+07}$	1.621e+07	2.115e+07	2.993e+07
14	$\boxed{1.147e+07}$	1.481e+07	2.519e+07	2.605e+07
15	$\boxed{1.026e+07}$	1.304e+07	2.413e+07	2.212e+07

Table I.6: Total constraints for constructing all Γ_n^a sets for the random POMDP problems with $|\mathcal{S}| = |\mathcal{Z}|$. T-test with $p = 0.95$.

Obs.	IpRr	IpNcs	TwoPass	Witness
3	$\boxed{1.126}$	$\boxed{0.983}$	$\boxed{0.992}$	1.718
4	$\boxed{2.622}$	$\boxed{2.516}$	$\boxed{2.123}$	4.354
5	$\boxed{6.139}$	$\boxed{6.620}$	$\boxed{4.847}$	12.819
6	$\boxed{11.220}$	$\boxed{12.772}$	$\boxed{9.042}$	25.860
7	28.087	38.388	$\boxed{22.463}$	100.498
8	$\boxed{38.697}$	55.178	$\boxed{30.261}$	135.446
9	$\boxed{66.162}$	104.318	$\boxed{52.009}$	342.910
10	$\boxed{109.606}$	169.262	$\boxed{77.995}$	457.261
11	$\boxed{264.084}$	$\boxed{371.350}$	$\boxed{184.213}$	722.968
12	$\boxed{379.709}$	596.626	$\boxed{237.799}$	1075.503
13	542.768	840.177	$\boxed{319.090}$	1241.831
14	$\boxed{732.718}$	987.782	$\boxed{488.382}$	1351.313
15	$\boxed{1077.523}$	1222.523	$\boxed{826.304}$	1455.357

Table I.7: Total execution time for constructing all Γ_n sets for the random POMDP problems with $|\mathcal{S}| = 7$. T-test with $p = 0.95$.

I.1.1 Total Running Time

Although we presented the running time for simply building the Γ_n^a sets, Tables I.7 through I.9 show the total mean running time for constructing Γ_n required by each of the algorithms on this data set. This include the additional time the PRUNE routine using to merge the set. Although this time is predominantly the same for the algorithms, using the PRUNE routine to merge the Γ_n^a into Γ_n means that the total number of constraints is sensitive to the order in which the vectors are processed.

States.	IpRr	IpNcs	TwoPass	Witness
3	3.036	2.902	1.316	2.488
4	6.618	6.972	3.615	8.783
5	9.804	10.790	6.198	17.053
6	11.401	12.897	7.430	21.602
7	28.087	38.388	22.463	100.498
8	50.155	77.846	42.457	231.338
9	66.903	98.801	58.880	308.017
10	90.424	140.467	82.312	422.917
11	194.620	331.244	176.911	720.640
12	355.842	486.314	339.195	954.094
13	407.308	584.145	388.002	1124.459
14	668.434	934.044	623.117	1254.969
15	878.698	1183.329	843.705	1602.123

Table I.8: Total execution time for constructing all Γ_n sets for the random POMDP problems with $|\mathcal{Z}| = 7$. T-test with $p = 0.95$.

States/Obs.	IpRr	IpNcs	TwoPass	Witness
3	0.474	$\boxed{0.409}$	$\boxed{0.337}$	0.490
4	1.343	1.227	$\boxed{0.856}$	1.497
5	3.758	3.768	$\boxed{2.613}$	5.442
6	7.672	8.496	$\boxed{5.570}$	14.013
7	28.087	38.388	$\boxed{22.463}$	100.498
8	$\boxed{100.680}$	164.842	$\boxed{79.480}$	548.893
9	$\boxed{448.249}$	725.270	$\boxed{359.523}$	1316.605
10	$\boxed{1056.990}$	1291.559	$\boxed{920.050}$	1573.244
11	$\boxed{1714.217}$	1755.611	$\boxed{1562.737}$	1803.747
12	$\boxed{1643.082}$	$\boxed{1665.157}$	$\boxed{1593.402}$	$\boxed{1691.941}$
13	$\boxed{1758.537}$	1820.684	$\boxed{1663.758}$	1803.529
14	1835.887	1821.383	1814.387	$\boxed{1803.511}$
15	$\boxed{1684.703}$	$\boxed{1674.622}$	$\boxed{1659.929}$	$\boxed{1673.424}$

Table I.9: Total execution time for constructing all Γ_n sets for the random POMDP problems with $|\mathcal{S}| = |\mathcal{Z}|$. T-test with $p = 0.95$.

I.2 Heuristic Algorithms

Section 6.7.5 discussed the results obtained by varying the entropy thresholds in the dual mode controllers. There the full table of results for the DM-MLS heuristic was presented. Here, Tables I.10 through I.12 show the results for the other dual mode controllers used in the emprical results.

Domain	Threshold 0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
4x3	0.971	1.665	1.731	1.760	1.739	1.760	1.732	1.726	1.754
4x4	0	0	0	0.104	**3.714**	**3.709**	**3.707**	**3.709**	**3.707**
CHEESE	3.417	3.424	3.428	3.417	3.436	3.440	3.440	3.433	3.443
PAINT	1.416	1.756	2.011	2.017	0.913	-8.515	**-8.479**	**-8.546**	**-8.488**
SHUTTLE	**32.702**	**32.627**	**32.688**	**32.658**	**32.782**	**32.725**	**32.672**	**32.634**	**32.552**
TIGER	15.897	19.353	19.745	-73.340	-74.279	-74.161	**-894.410**	**-897.218**	**-893.218**
NETWORK	-594.924	-595.161	-436.487	-314.882	-225.196	-47.685	18.387	39.504	42.827
NONLIN	6.694	6.678	6.682	6.655	6.685	6.682	**6.311**	**6.282**	**6.284**
SACI	-80.657	-52.901	-48.926	**-31.757**	**-32.067**	**-31.939**	**-32.914**	**-32.345**	**-32.088**
HALLWAY	0.470	0.527	0.593	0.605	0.823	0.815	0.812	0.804	0.804
HALL.-2	0.013	0.014	0.012	0.024	0.153	0.206	0.213	0.215	**0.176**
CIT	0.028	0.348	0.306	0.197	0.687	0.796	0.805	**0.806**	**0.807**
MIT	-2.465	-0.197	-0.020	0.270	0.816	0.851	0.854	**0.859**	**0.857**
SUNY.	-42.653	-41.498	-8.017	-1.377	-0.112	0.677	0.751	**0.764**	**0.766**
PENT.	-4.765	-2.646	-0.365	0.115	0.693	0.780	0.785	**0.795**	**0.793**
FOURTH	-3.138	-1.376	-0.419	-0.203	0.220	0.558	0.573	**0.585**	**0.588**
IFF	3.677	6.178	5.932	7.677	8.238	**8.071**	**8.194**	**8.347**	**8.300**
BB	0.468	0.558	0.572	0.623	0.637	0.628	**0.632**	**0.644**	**0.623**
MACHINE	-123.617	-33.045	1.667	17.018	20.458	49.620	56.875	**57.063**	**57.395**
ALOHA10	90.217	92.489	97.725	102.932	106.940	112.516	116.099	119.006	124.652
ALOHA30	679.141	684.598	689.384	694.387	702.627	728.664	770.127	833.304	**850.332**
CIT-U	-27.744	-27.275	-27.222	-24.151	-7.825	-0.827	-0.760	-0.311	**0.625**
MIT-U	-31.144	-30.988	-30.383	-29.963	-14.465	-2.553	-0.631	-0.337	**0.531**
SUNY.-U	-20.806	-19.940	-15.771	-15.172	-10.252	-0.225	0.180	0.492	0.490
PENT.-U	-28.153	-27.896	-27.448	-24.072	-4.804	-0.703	-0.656	-0.238	0.692
FOURTH-U	-26.836	-25.355	-24.234	-23.703	-12.214	-0.136	0.054	**0.451**	**0.456**

Table I.10: Threshold values and the ADM-MLS heuristic. T-test with $p = 0.995$.

Domain	**Threshold** 0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
4x3	1.387	1.431	1.503	1.514	1.774	1.812	1.796	1.790	1.790
4x4	3.491	3.485	3.492	3.491	3.494	3.491	3.651	3.659	3.708
CHEESE	3.213	3.207	3.460	3.465	3.466	3.461	3.462	3.465	3.460
PAINT	1.643	2.128	2.778	2.788	3.127	2.299	**2.261**	**2.304**	**2.291**
SHUTTLE	**32.550**	**32.639**	**32.655**	**32.743**	**32.650**	**32.663**	**32.667**	**32.691**	**32.783**
TIGER	16.192	19.470	19.258	19.651	19.197	19.292	19.221	18.494	19.690
NETWORK	-595.332	-595.064	-595.218	-595.097	-595.177	-435.335	-410.796	-235.787	187.911
NONLIN	6.669	6.688	6.680	**6.300**	**6.295**	**6.329**	**6.277**	**6.279**	**6.255**
SACI	-81.677	-81.525	-81.781	-80.996	-80.693	-80.639	-77.985	-65.096	-56.146
HALLWAY	0.258	0.489	0.528	0.598	0.466	0.355	0.348	0.344	0.356
HALL.-2	0.014	0.080	0.164	0.193	0.171	0.142	0.130	0.124	0.122
CIT	-58.072	-0.175	0.759	0.826	**0.833**	**0.832**	**0.834**	**0.833**	**0.832**
MIT	-0.172	0.802	0.815	0.809	0.810	0.810	0.808	0.814	0.811
SUNY.	-9.166	0.482	0.743	**0.760**	**0.759**	**0.758**	**0.759**	**0.760**	**0.758**
PENT.	-7.051	0.117	0.717	0.804	**0.821**	**0.821**	**0.821**	**0.821**	**0.821**
FOURTH	-9.686	0.327	0.584	**0.594**	**0.592**	**0.590**	**0.591**	**0.592**	**0.592**
IFF	-3.101	-2.974	-2.073	-1.313	2.323	4.552	**4.419**	**4.532**	**5.004**
BB	0.350	0.322	**0.094**	**0.101**	**0.099**	**0.100**	**0.099**	**0.093**	**0.103**
MACHINE	-408.124	-286.247	-124.016	-41.201	5.903	21.258	33.379	52.097	**59.694**
ALOHA10	92.184	97.971	104.935	112.146	119.541	126.803	**126.311**	**128.061**	**127.217**
ALOHA30	686.231	687.099	696.308	711.341	754.577	839.081	**846.941**	**850.651**	**849.892**
CIT-U	-26.730	-15.061	-6.567	-0.233	-0.160	0.063	-0.747	0.358	0.370
MIT-U	-22.904	-13.267	-6.576	-5.842	0.345	0.408	0.543	0.556	0.554
SUNY.-U	-26.827	-19.301	-14.816	-11.970	-4.979	0.090	0.305	0.328	0.330
PENT.-U	-27.536	-26.804	-15.377	-1.797	-0.424	-0.766	-1.395	0.575	0.580
FOURTH-U	-36.943	-17.536	-10.910	-5.625	-3.784	-0.599	-0.966	0.338	0.347

Table I.11: Threshold values and the DM-QMDP heuristic. T-test with $p = 0.995$.

Domain	Threshold 0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
4x3	0.947	1.680	1.718	1.763	1.807	1.812	**1.863**	**1.863**	**1.879**
4x4	0	0	0	0.222	**3.711**	**3.713**	**3.707**	**3.711**	**3.714**
CHEESE	3.423	3.428	3.423	3.422	3.463	3.466	3.467	3.459	3.465
PAINT	1.416	1.755	2.017	2.023	0.873	**-0.477**	**-0.447**	**-0.483**	**-0.487**
SHUTTLE	**32.673**	**32.609**	**32.685**	**32.584**	**32.643**	**32.625**	**32.701**	**32.676**	**32.710**
TIGER	16.250	19.253	19.636	19.547	18.858	18.893	18.600	18.919	19.351
NETWORK	-595.253	-595.001	-436.661	-313.824	-224.401	-48.206	19.763	187.312	187.746
NONLIN	6.665	6.658	6.667	6.672	6.686	6.673	**6.283**	**6.289**	**6.302**
SACI	-81.432	-72.988	-68.831	**7.457**	**7.713**	**7.548**	**7.651**	**7.583**	**7.562**
HALLWAY	0.465	0.521	0.602	0.540	0.515	0.382	0.357	0.351	**0.345**
HALL.-2	0.016	0.014	0.013	0.026	0.179	0.224	0.140	0.132	**0.103**
CIT	0.049	0.367	0.328	0.234	0.701	0.821	0.827	**0.833**	**0.832**
MIT	-2.027	0.019	0.043	0.304	0.763	0.802	0.808	**0.809**	**0.810**
SUNY.	-42.626	-42.075	-7.967	-1.441	-0.251	0.627	0.686	**0.758**	**0.757**
PENT.	-5.056	-2.345	-0.745	-0.205	0.647	0.803	0.811	**0.822**	**0.821**
FOURTH	-8.650	-5.874	-1.176	-0.217	0.331	0.574	0.591	**0.590**	**0.592**
IFF	3.067	4.415	5.684	5.435	5.654	**4.641**	**4.613**	**4.873**	**4.369**
BB	0.461	0.240	0.227	0.127	0.117	0.119	**0.097**	**0.100**	**0.097**
MACHINE	-123.453	-33.170	1.882	17.058	20.265	51.112	**59.354**	**59.725**	**59.460**
ALOHA10	89.948	92.684	96.829	102.581	106.651	112.249	117.271	118.960	125.947
ALOHA30	680.278	684.565	687.499	695.593	704.424	731.142	770.233	840.747	**847.335**
CIT-U	-27.746	-27.146	-26.907	-24.644	-7.916	-0.894	-0.816	-0.563	**0.369**
MIT-U	-31.223	-30.725	-30.362	-30.286	-14.900	-4.204	-0.620	-0.402	**0.553**
SUNY.-U	-20.595	-19.940	-15.764	-14.995	-11.849	-0.202	0.053	0.320	**0.326**
PENT.-U	-28.260	-27.932	-27.434	-24.013	-5.002	-0.679	-0.656	-0.380	0.570
FOURTH-U	-26.839	-25.371	-23.910	-23.693	-12.434	-0.183	-0.004	**0.347**	**0.341**

Table I.12: Threshold values and the ADM-QMDP heuristic. T-test with $p = 0.995$.

We presented the results for various entropy thresholds for the dual mode controllers, DM-MLS, ADM-MLS, DM-Q-MDP and ADM-Q-MDP in Section 6.7.5 and used a black background to indicate settings for which the entropy never exceeded that level. Tables I.13 through I.16 show the percentage of entropy actions taken for the dual mode controllers; i.e., the number of steps for which the information state entropy exceeded the threshold.

Domain	Threshold 0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
4x3	34%	31%	22%	21%	15%	1%	1%	1%	1%
4x4	77%	77%	67%	53%	53%	53%	38%	38%	20%
CHEESE	26%	26%	6%	6%	1%	1%	1%	1%	1%
PAINT	85%	80%	72%	72%	50%	33%	33%	0%	0%
SHUTTLE	0%	0%	0%	0%	0%	0%	0%	0%	0%
TIGER	81%	73%	73%	73%	73%	73%	50%	50%	50%
NETWORK	50%	50%	50%	50%	50%	34%	32%	22%	1%
NONLIN	27%	27%	27%	0%	0%	0%	0%	0%	0%
SACI	82%	77%	75%	66%	64%	64%	25%	4%	4%
HALLWAY	93%	81%	76%	31%	21%	7%	3%	1%	1%
HALL.-2	100%	88%	62%	23%	14%	8%	2%	1%	1%
CIT	95%	9%	1%	0%	0%	0%	0%	0%	0%
MIT	20%	2%	0%	0%	0%	0%	0%	0%	0%
SUNY.	50%	6%	0%	0%	0%	0%	0%	0%	0%
PENT.	43%	9%	2%	1%	0%	0%	0%	0%	0%
FOURTH	82%	3%	0%	0%	0%	0%	0%	0%	0%
IFF	71%	28%	9%	1%	0%	0%	0%	0%	0%
BB	100%	95%	0%	0%	0%	0%	0%	0%	0%
MACHINE	33%	25%	14%	9%	6%	4%	4%	1%	0%
ALOHA10	53%	40%	26%	11%	5%	0%	0%	0%	0%
ALOHA30	69%	54%	40%	34%	25%	4%	0%	0%	0%
CIT-U	95%	81%	58%	19%	16%	13%	14%	4%	3%
MIT-U	93%	82%	63%	58%	11%	8%	5%	3%	3%
SUNY.-U	94%	83%	72%	61%	43%	6%	4%	2%	2%
PENT.-U	96%	94%	77%	32%	28%	30%	34%	6%	4%
FOURTH-U	95%	66%	51%	36%	28%	10%	10%	3%	2%

Table I.13: Percentage of entropy reduction actions taken for the DM-MLS heuristic.

Domain	Threshold 0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
4x3	45%	28%	24%	21%	1%	1%	0%	0%	0%
4x4	50%	50%	50%	50%	0%	0%	0%	0%	0%
CHEESE	10%	10%	10%	10%	5%	1%	1%	1%	1%
PAINT	85%	80%	72%	72%	50%	33%	0%	0%	0%
SHUTTLE	0%	0%	0%	0%	0%	0%	0%	0%	0%
TIGER	81%	73%	73%	50%	50%	50%	0%	0%	0%
NETWORK	50%	50%	34%	26%	21%	14%	11%	1%	1%
NONLIN	27%	27%	27%	27%	27%	27%	0%	0%	0%
SACI	56%	11%	5%	0%	0%	0%	0%	0%	0%
HALLWAY	81%	78%	71%	68%	11%	5%	2%	2%	0%
HALL.-2	100%	100%	100%	98%	29%	12%	4%	2%	0%
CIT	15%	10%	9%	12%	4%	2%	1%	0%	0%
MIT	37%	21%	20%	15%	5%	1%	1%	0%	0%
SUNY.	80%	79%	43%	22%	13%	3%	1%	0%	0%
PENT.	49%	41%	20%	15%	5%	2%	1%	0%	0%
FOURTH	15%	11%	7%	6%	4%	1%	1%	0%	0%
IFF	54%	38%	38%	17%	15%	0%	0%	0%	0%
BB	23%	16%	16%	2%	1%	1%	0%	0%	0%
MACHINE	14%	8%	6%	5%	5%	2%	0%	0%	0%
ALOHA10	61%	52%	41%	25%	18%	12%	8%	5%	1%
ALOHA30	70%	62%	47%	40%	36%	30%	21%	4%	0%
CIT-U	94%	94%	94%	89%	71%	45%	43%	3%	0%
MIT-U	99%	98%	98%	98%	69%	44%	38%	3%	0%
SUNY.-U	83%	82%	78%	76%	57%	40%	31%	3%	0%
PENT.-U	98%	98%	98%	96%	71%	47%	46%	4%	0%
FOURTH-U	94%	92%	90%	89%	62%	45%	41%	0%	0%

Table I.14: Percentage of entropy reduction actions taken for the ADM-MLS heuristic.

Domain	Threshold 0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
4x3	34%	31%	22%	21%	16%	1%	1%	1%	1%
4x4	77%	77%	67%	53%	53%	53%	38%	38%	20%
CHEESE	26%	26%	6%	6%	1%	1%	1%	1%	1%
PAINT	85%	80%	72%	72%	50%	12%	0%	0%	0%
SHUTTLE	0%	0%	0%	0%	0%	0%	0%	0%	0%
TIGER	81%	73%	73%	73%	73%	73%	37%	37%	37%
NETWORK	50%	50%	50%	50%	50%	34%	32%	22%	1%
NONLIN	27%	27%	27%	0%	0%	0%	0%	0%	0%
SACI	82%	80%	76%	66%	63%	62%	53%	26%	16%
HALLWAY	93%	81%	75%	18%	10%	3%	2%	1%	1%
HALL.-2	100%	80%	52%	11%	7%	5%	2%	1%	1%
CIT	95%	10%	1%	0%	0%	0%	0%	0%	0%
MIT	20%	2%	0%	0%	0%	0%	0%	0%	0%
SUNY.	48%	4%	1%	0%	0%	0%	0%	0%	0%
PENT.	49%	10%	2%	0%	0%	0%	0%	0%	0%
FOURTH	34%	2%	0%	0%	0%	0%	0%	0%	0%
IFF	78%	58%	38%	20%	6%	1%	0%	0%	0%
BB	100%	46%	0%	0%	0%	0%	0%	0%	0%
MACHINE	33%	25%	14%	9%	6%	4%	4%	2%	0%
ALOHA10	53%	40%	26%	12%	6%	0%	0%	0%	0%
ALOHA30	69%	54%	40%	34%	25%	4%	0%	0%	0%
CIT-U	95%	73%	45%	12%	9%	7%	10%	2%	2%
MIT-U	93%	82%	53%	44%	7%	6%	4%	2%	2%
SUNY.-U	94%	77%	58%	44%	27%	4%	2%	2%	1%
PENT.-U	97%	94%	77%	31%	19%	18%	20%	3%	2%
FOURTH-U	94%	64%	47%	32%	24%	10%	10%	3%	1%

Table I.15: Percentage of entropy reduction actions taken for the DM-QMDP heuristic.

Domain	Threshold 0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
4x3	45%	28%	24%	21%	1%	1%	0%	0%	0%
4x4	50%	50%	50%	49%	0%	0%	0%	0%	0%
CHEESE	10%	10%	10%	10%	5%	1%	1%	1%	1%
PAINT	85%	80%	72%	72%	50%	0%	0%	0%	0%
SHUTTLE	0%	0%	0%	0%	0%	0%	0%	0%	0%
TIGER	81%	73%	73%	37%	37%	37%	0%	0%	0%
NETWORK	50%	50%	34%	26%	21%	14%	11%	1%	1%
NONLIN	27%	27%	27%	27%	27%	27%	0%	0%	0%
SACI	62%	41%	32%	0%	0%	0%	0%	0%	0%
HALLWAY	81%	78%	71%	56%	4%	2%	1%	1%	0%
HALL.-2	100%	100%	100%	98%	16%	5%	2%	1%	0%
CIT	15%	10%	9%	11%	4%	2%	1%	0%	0%
MIT	34%	17%	16%	12%	4%	1%	0%	0%	0%
SUNY.	80%	79%	42%	22%	14%	3%	2%	0%	0%
PENT.	50%	39%	24%	18%	6%	2%	1%	0%	0%
FOURTH	32%	26%	12%	7%	3%	1%	0%	0%	0%
IFF	47%	34%	29%	17%	10%	0%	0%	0%	0%
BB	15%	1%	1%	0%	0%	0%	0%	0%	0%
MACHINE	14%	8%	6%	5%	5%	1%	0%	0%	0%
ALOHA10	61%	52%	41%	25%	18%	12%	8%	5%	1%
ALOHA30	70%	62%	47%	40%	36%	30%	22%	4%	0%
CIT-U	94%	94%	94%	90%	71%	42%	40%	2%	0%
MIT-U	99%	98%	98%	97%	65%	43%	31%	2%	0%
SUNY.-U	83%	82%	77%	76%	59%	41%	29%	2%	0%
PENT.-U	99%	98%	98%	96%	71%	46%	45%	2%	0%
FOURTH-U	94%	92%	89%	89%	62%	45%	40%	0%	0%

Table I.16: Percentage of entropy reduction actions taken for the ADM-QMDP heuristic.

Bibliography

[1] M. Aoki. *Optimization of Stochastic Systems.* Academic Press, New York, NY, 1967.

[2] K. J. Astrom. Optimal control of Markov decision processes with incomplete state estimation. *Journal of Mathematical Analysis and Applications*, 10:174–205, 1965.

[3] K. J. Astrom. Optimal control of Markov decision processes with incomplete state estimation II. *Journal of Mathematical Analysis and Applications*, 26:403–406, 1969.

[4] K. J. Astrom. Theory and applications of adaptive control — A survey. *Automatica*, 19:471–486, 1983.

[5] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning: Proceedings of the Twelfth International Conference*, pages 30–37, San Francisco, CA, 1995. Morgan Kaufmann.

[6] Andrew G. Barto, Richard S. Sutton, and Christopher J. C. H. Watkins. Learning and sequential decision making. In M. Gabriel

and J.W. Moore, editors, *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pages 539–602. MIT Press, Cambridge, Massachusetts, 1990.

[7] Richard Bellman. *Dynamic Programming.* Princeton University Press, Princeton, New Jersey, 1957.

[8] Dimitri P. Bertsekas. Distributed dynamic programming. *IEEE Transactions on Automatic Control*, AC-27:610–616, 1982.

[9] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control, Vols. 1 and 2.* Athena Scientific., Belmont, Massachusetts, 1995.

[10] Dimitri P. Bertsekas and R. G. Gallagher. *Data Networks.* Prentice Hall., Englewood Cliffs, N.J., 1992.

[11] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific., Belmont, Massachusetts, 1996.

[12] David Blackwell. Discrete dynamic programming. *Annals of Mathematical Statistics*, 33(2):719–726, June 1962.

[13] David Blackwell. Discounted dynamic programming. *Annals of Mathematical Statistics*, 36:226–235, 1965.

[14] Craig Boutilier, Richard Dearden, and Moises Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Montreal, Canada, 1995.

[15] Craig Boutilier and David Poole. Computing optimal policies for partially observable decision processes using compact representations. In

Proceedings of the Thirteenth National Conference on Artificial Intelligence, pages 1168–1175, Portland, Oregon, 1996.

[16] Ronen I. Brafman. A heuristic variable grid solution method for POMDPs. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 727–733, Providence, Rhode Island, 1997.

[17] William L. Briggs. *A multigrid tutorial.* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1987.

[18] J. Buhmann, W. Burgard, Cremers A., D. Fox, T. Hofmann, F. Scheider, J. Strikos, and S. Thrun. The mobile robot RHINO. *AI Magazine*, 16(2):31–37, Summer 1995.

[19] Bruce Bukiet, Elliotte Rusty Harold, and Jose Luis Palacios. A Markov chain approach to baseball. *Operations Research*, 45(1):14–23, 1997.

[20] Dima Burago, Michel de Rougemont, and Anatol Slissenko. On the complexity of partially observed Markov decision processes. *Theoretical Computer Science*, 157(2):161–183, 1996.

[21] Anthony Cassandra, Leslie Kaelbling, and James Kurien. Acting under uncertainty: Discrete bayesian models for mobile-robot navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1996.

[22] Anthony R. Cassandra. Algorithms for partially observable Markov decision processes. Technical Report CS-94-14, Brown University, Providence, Rhode Island, 1994.

[23] Anthony R. Cassandra, Leslie Pack Kaelbling, and Michael L. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1023–1028, Seattle, Washington, 1994.

[24] Anthony R. Cassandra, Michael L. Littman, and Nevin L. Zhang. Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–97)*, Providence, Rhode Island, 1997.

[25] David A. Castanon. Approximate dynamic programming for sensor management. In *Proceedings of the Conference on Decision and Control* To appear, San Diego, CA, 1997.

[26] Hsien-Te Cheng. *Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, University of British Columbia, British Columbia, Canada, 1988.

[27] Hsien-Te Cheng. Personal communication, 1994.

[28] Lonnie Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 183–188, San Jose, California, 1992. AAAI Press.

[29] Gregg Collins and Louise Pryor. Achieving the functionality of filter conditions in a partial order planner. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 375–380, 1992.

[30] Anne Condon, Joan Feigenbaum, Carsten Lund, and Peter Shor. Probabilistic checkable debate systems and nonapproximability of PSPACE-hard functions. *Chicago Journal of Theoretical Computer Science*, (4), 1995.

[31] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[32] Robert Harry Crites. *Large-scale Dynamic Optimization Using Teams of Reinforcement Learning Agents*. PhD thesis, University of Massachusetts, September 1996.

[33] Bruce D'Ambrosio and Robert Fung. Far sighted approaches to sensor management experiments in reinforcement learning. Technical report, Prevision, 1996.

[34] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning with deadlines in stochastic domains. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, DC, 1993.

[35] Thomas L. Dean, Robert Givan, and Sonia M. Leach. Model reduction techniques for computing aproximately optimal solutions for Markov decision processes. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–97)*, pages 124–131, Providence, Rhode Island, 1997.

[36] M. DeGroot. *Optimal Statistical Decisions.* McGraw-Hill, New York, N.Y., 1970.

[37] Denise Draper, Steve Hanks, and Daniel Weld. Probabilistic planning with information gathering and contingent execution. Technical Report 93-12-04, University of Washington, December 1993.

[38] E. B. Dynkin. Controlled random sequences. *Theory of Probability and its Applications*, X:1–14, 1965.

[39] James N. Eagle. The optimal search for a moving target when the search path is constrained. *Operations Research*, 32(5):1107–1115, 1984.

[40] James E. Eckles. Optimum maintenance with incomplete information. *Operations Research*, 16:1058–1067, 1968.

[41] Hugh Ellis, Mingxiang Jiang, and Ross B. Corotis. Inspection, maintenance, and repair with partial observability. *Journal of Infrastructure Systems*, 1(2):92–99, 1995.

[42] Adriane V. Gheorghe. *Decision Processes in Dynamic Probabilistic Systems.* Kluwer Academic Publishers., Norwell, MA., 1990.

[43] Robert Givan. Personal communication, 1996.

[44] Judy Goldsmith, Chris Lusena, and Martin Mundhenk. The complexity of deterministically observable finite-horizon Markov decision processes. Technical Report 268-96, University of Kentucky, Lexington, Kentucky, December 1996.

[45] Eric A. Hansen. An improved policy iteration algorithm for partially observable MDPs. *NIPS*, 1997.

[46] Milos Hauskrecht. Incremental methods for computing bounds in partially observable Markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 734–739, Providence, Rhode Island, 1997.

[47] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

[48] Howard and Matheson. Risk sensitive Markov decision processes. *Management Science*, 18(7):356–370, 1972.

[49] Ronald A. Howard. *Dynamic Programming and Markov Processes.* The MIT Press, Cambridge, Massachusetts, 1960.

[50] J. Hughes. Optimal internal audit timing. *Accounting Review*, LII:56–58, 1977.

[51] Tommi Jaakkola, Satinder P. Singh, and Michael I. Jordan. Monte-carlo reinforcement learning in non-Markovian decision problems. In *Advances in Neural Information Processing Systems 7*, 1995.

[52] Leslie Pack Kaelbling, Michael L. Littman, and Anthony Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence* To appear, 1998.

[53] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research (JAIR)*, 4, 1996.

[54] J. S. Kakalik. Optimal policies for partially observable Markov systems. Technical Report TR-18, Massachusetts Institute of Technology, Cambridge, MA., October 1965.

[55] R. E. Kalman. A new apporach to linear filtering and prediction problems. *Journal of Basic Engineering*, pages 35–45, March 1960.

[56] R. Kaplan. Optimal investigation strategies with imperfect information. *Journal of Accounting Research*, 7:32–43, 1969.

[57] W. Karush and R. Dear. Optimal strategy for item presentation in learning models. *Management Science*, 13:773–785, 1967.

[58] Sven Koenig. Optimal probabilistic and decision-theoretic planning using Markovian decision theory. Technical Report UCB/CSD 92/685, Berkeley, May 1992.

[59] Sven Koenig and Reid Simmons. Unsupervised learning of probabilistic models for robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1996.

[60] P. R. Kumar. A survey of some results in stochastic adaptive control. *SIAM Journal on Control and Optimization*, 23:329–380, 1985.

[61] Nicholas Kushmeric, Steve Hanks, and Daniel Weld. An algorithm for probabilistic planning. Technical Report 93-06-03, Department of Computer Science, University of Washington, 1993.

[62] Nicholas Kushmerick, Steve Hanks, and Daniel S. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1-2):239–286, September 1995.

[63] Harold J. Kushner and A. J. Kleinman. Mathematical programming and the control of Markov chains. *International Journal of Control*, 13(5):801–820, 1971.

[64] Daniel E. Lane. A partially observable model of decision making by fishermen. *Operations Research*, 37:240, 1989.

[65] J.J. Leonard and Hugh Durrant-Whyte. Localization by tracking geometric beacons. *IEEE Transactions on Robotics and Automation*, 7(6), 1991.

[66] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[67] Long-Ji Lin and Tom M. Mitchell. Memory approaches to reinforcement learning in non-Markovian domains. Technical Report CMU-CS-92-138, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1992.

[68] Michael Littman, Anthony Cassandra, and Leslie Kaelbling. Learning policies for partially observable environments: Scaling up. In *Machine Learning: Proceedings of the Twelfth International Conference*, pages 362–370, San Francisco, CA, 1995. Morgan Kaufmann.

[69] Michael L. Littman. Memoryless policies: Theoretical limitations and practical results. In *From Animals to Animats 3*, Brighton, UK, 1994.

[70] Michael L. Littman. The witness algorithm for solving partially observable Markov decision processes. Technical Report CS-94-40, Brown University, Providence, Rhode Island, 1994.

[71] Michael L. Littman. Personal communication, 1996.

[72] Michael L. Littman. *Algorithms for Sequential Decision Making*. PhD thesis, Department of Computer Science, Brown University, February 1996. Also Technical Report CS-96-09.

[73] Michael L. Littman. Probabilistic propositional planning: Representations and complexity. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 748–754, Providence, Rhode Island, 1997.

[74] Michael L. Littman, Anthony R. Cassandra, and Leslie Pack Kaelbling. Efficient dynamic-programming updates in partially observable Markov decision processes. Technical Report CS-95-19, Brown University, Providence, Rhode Island, 1995.

[75] William S. Lovejoy. Computationally feasible bounds for partially observed Markov decision processes. *Operations Research*, 39(1):162–175, 1991.

[76] William S. Lovejoy. A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, 28(1):47–65, 1991.

[77] Marc Mangel and Colin W. Clark. *Dynamic Modeling in Behavioral Ecology.* Princeton University Press, Princeton, New Jersey, 1988.

[78] A. Manne. Linear programming and sequential decisions. *Management Science*, 6:259–267, 1960.

[79] T. M. Mansell. A method for planning given uncertain and incomplete information. In *Proceedings of the 9th Conference on Uncertainty in Artificial Intelligence*, pages 350–358. Morgan Kaufmann Publishers, July 1993.

[80] T. H. Mattheis. An algorithm for determining irrelevant constraints and all verticies in systems of linear inequalities. *Operations Research*, 21:247–260, 1973.

[81] T. H. Mattheis and David S. Rubin. A survey and comparison of methods for finding all vertices of convex polyhedral sets. *Mathematics of Operations Research*, 5(2):167–185, 1980.

[82] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the 9th National Conference on Artificial Intelligence*, 1991.

[83] Andrew Kachites McCallum. Efficient exploration in reinforcement learning with hidden state. Technical report, University of Rochester, Rochester, New York, 1996.

[84] Andrew Kachites McCallum. *Reinforcement Learning with Selective Perception and Hidden State.* PhD thesis, University of Rochester, 1996.

[85] R. Andrew McCallum. Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Conference on Machine Learning*, Amherst, Massachusetts, 1993. Morgan Kaufmann.

[86] R. Andrew McCallum. Instance-based utile distinctions for reinforcement learning with hidden state. In *Proceedings of the Twelfth International Conference Machine Learning*, pages 387–395, San Francisco, CA, 1995. Morgan Kaufmann.

[87] George E. Monahan. A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science*, 28(1):1–16, 1982.

[88] Leora Morgenstern. Knowledge preconditions for actions and plans. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 867–874, 1987.

[89] Sraban Mukherjee and Kiran Seth. A corrected and improved computational scheme for partially observable Markov processes. *INFOR*, 29(3):206–212, 1991.

[90] Martin Mundhenk, Judy Goldsmith, and Eric Allender. The complexity of unobservable finite-horizon Markov decision processes. Technical Report 269-96, University of Kentucky, Lexington, Kentucky, December 1996.

[91] Martin Mundhenk, Judy Goldsmith, and Eric Allender. The complexity of policy evaluation for finite-horizon partially-observable markov decision processes. In *Proceedings of the 25th Mathematical Foundations of Computer Sciences*, pages 129–138. Lecture Notes in Computer Science #1295, Springer-Verlag, 1997.

[92] Martin Mundhenk, Judy Goldsmith, Chris Lusena, and Eric Allender. Encyclopaedia of complexity results for finite-horizon Markov decision process problems. Technical Report TR 273-97, University of Kentucky, Lexington, Kentucky, September 1997.

[93] Remi Munos. A convergent reinforcement learning algorithm in the continuous case: the finite-element reinforcement learning. In *Proceedings of the Thirteenth International Conference on Machine Learning*, 1996.

[94] Illah Nourbakhsh, Rob Powers, and Stan Birchfield. Dervish: An office-navigating robot. *AI Magazine*, pages 53–60, Summer 1995.

[95] Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

[96] Ronald Parr and Stuart Russell. Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1088–1094. Morgan Kaufmann, 1995.

[97] J. S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the third international conference on principles of knowledge representation and reasoning*, pages 103–114, 1992.

[98] Mark A. Peot and David E. Smith. Conditional nonlinear planning. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 189–197, 1992.

[99] W. Pierskalla and J. Voelker. A survey of maintenance models: The control and surveillance of deteriorating systems. *Naval Research Logistics Quarterly*, 23:353–388, 1976.

[100] Loren K. Platzman. Optimal infinite-horizon undiscounted control of finite probabilistc systems. *SIAM Journal of Control and Optimization*, 18:362–380, 1980.

[101] Pollock. A simple model of search for a moving target. *Operations Research*, 18:883–903, 1970.

[102] Martin L. Puterman. *Markov Decision Processes — Discrete Stochastic Dynamic Programming.* John Wiley & Sons, Inc., New York, New York, 1994.

[103] Martin L. Puterman and Moon Chirl Shin. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24:1127–1137, 1978.

[104] Mark B. Ring. *Continual Learning in Reinforcement Environments.* PhD thesis, University of Texas, Austin, 1994.

[105] Donald Rosenfeld. Markovian deterioration with uncertain information. *Operations Research*, 24(1):141–155, 1976.

[106] Sheldon M. Ross. Quality control under Markovian deterioration. *Management Science*, 17(9):587596, 1971.

[107] Ulrich Rüde. *Mathematical and computational techniques for multilevel adaptive methods.* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1993.

[108] Katsushige Sawaki and Akira Ichikawa. Optimal control for partially observable Markov decision processes over an infinite horizon. *Journal of the Operations Research Society of Japan*, 21(1):1–14, March 1978.

[109] Y. Sawaragi and T. Yoshikawa. Discrete time Markov decision processes with incomplete state information. *Annals of Mathematics and Statistics*, 41:78–86, 1970.

[110] Jurgen Schmidhuber. Reinforcement learning in Markovian and non-Markovian environments. In *Advances in Neural Information Processing Systems 3*, pages 500–506, 1991.

[111] A. Segall. Dynamic file assignment in a computer network. *IEEE Transactions on Automatic Control*, AC-21:161–173, 1976.

[112] Hagit Shatkay and Leslie Pack Kaelbling. Learning topological maps with weak local odometric information. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, Nagoya, Japan, August 1997.

[113] Reid Simmons and Sven Koenig. Probabilistic navigation in partially observable environments. In *Fourteenth International Joint Conference on Artificial Intelligence*, pages 1080–1087, Montreal, Canada, 1995. Morgan Kaufmann.

[114] Richard Smallwood. The analysis of economic teaching strategies for a simple learning model. *Journal of Math Psych*, 8:285–301, 1971.

[115] Richard Smallwood, Edward Sondik, and F. Offensend. Toward and integrated methodology for the analysis of health-care systems. *Operations Research*, 19:1300–1322, 1971.

[116] Richard D. Smallwood and Edward J. Sondik. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, 21:1071–1088, 1973.

[117] Edward J. Sondik. *The Optimal Control of Partially Observable Markov Processes*. PhD thesis, Stanford University, Stanford, California, 1971.

[118] Edward J. Sondik. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research*, 26(2):282–304, 1978.

[119] Edward J. Sondik. Personal communication, 1994.

[120] C. T. Striebel. Sufficient statistics in the optimal control of stochastic systems. *Journal of Mathematical Analysis and Applications*, 12:576–592, 1965.

[121] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[122] G. J. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[123] Sylvie Thiebeaux, Marie-Odile Cordier, Olivier Jehl, and Jean-Paul Krivine. Supply restoration in power distribution systems — a case study in integrating model-based diagnosis and repair planning. In *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI–96)*, pages 525–532, Portland, Oregon, 1996.

[124] J. van Leeuwen, editor. *Algorithms and Complexity*. Elsevier Science Publishers, 1990.

[125] Rich Washington. Uncertainty and real-time therapy planning: incremental Markov-model approaches. *AAAI Spring Symposium on Artificial Intelligence in Medicine*, 1996.

[126] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[127] Chelsea C. White, III. Cost equality and inequality results for a partially observed stochastic optimization problem. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-5(6):576–582, November 1975.

[128] Chelsea C. White, III. Optimal diagnostic questionaires which allow less than truthful responses. *Information and Control*, 32:61–74, 1976.

[129] Chelsea C. White, III. Procedures for the solution of a finite-horizon partially observed, semi-Markov optimization problem. *Operations Research*, 24(2):348–358, 1976.

[130] Chelsea C. White, III. Monotone control laws for noisy, countable-state Markov chains. *European Journal of Operations Research*, 5:124–132, 1980.

[131] Chelsea C. White, III. Partially observed Markov decision processes: A survey. *Annals of Operations Research*, 32, 1991.

[132] Chelsea C. White, III and William T. Scherer. Solution procedures for partially observed Markov decision processes. *Operations Research*, 37(5):791–797, 1989.

[133] Chelsea C. White, III and William T. Scherer. Finite memory suboptimal design for partially observed Markov decision processes. *Operations Research*, 42(3):439, 455.

[134] Steven D. Whitehead and Dana H. Ballard. Learning to perceive and act by trial and error. *Machine Learning*, 7(1):45–83, 1991.

[135] Marco Wiering and Jurgen Schmidhuber. HQ-learning: discovering Markovian subgoals for non-Markovian reinforcement learning. Technical Report IDSIA-95-96, IDSIA, Switzerland, October 1996.

[136] David Wilkins, Karen Myers, John Lowrance, and Leonard Wesley. Planning and reacting in uncertain and dynamic environments. *J. Expt. Theor. Artificial Intelligence*, 7:121–152, 1995.

[137] Wayne L. Winston. *Introduction to Mathematical Programming: Applications and Algorithms*. PWS-KENT, Boston, Massachusetts, 1991.

[138] Nevin L. Zhang. Efficient planning in stochastic domains through exploiting problem characteristics. Technical Report HKUST-CS95-40, Department of Computer Science, Hong Kong University of Science and Technology, August 1995.

[139] Nevin L. Zhang. Personal communication, 1997.

[140] Nevin L. Zhang and Wenju Liu. Planning in stochastic domains: Problem characteristics and approximation. Technical Report HKUST-CS96-31, Department of Computer Science, Hong Kong University of Science and Technology, 1996.

[141] Nevin L. Zhang and Wenju Liu. Region-based approximations for planning in stochastic domains. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–97)*, pages 472–480, Providence, Rhode Island, 1997.

Notation

Symbols

$\mathcal{A}$	The set of actions or decision choices in a process.
A^t	The action chosen at time t.
a, a'	A particular action in the set $\mathcal{A}$
AH	Expected action entropy. (Equation 6.7)
$\mathcal{B}$	Space of information states, or space of probabilities distributions over $\mathcal{S}$.
$\hat{\mathcal{B}}$	A finite set of information states, such that each leads to a seperate linear segment in the parsimonious representation of a value function.
$\mathcal{B}'(b, a)$	The set of succesor information states of the state b under action a.
b	An information or belief state, which is a probability distribution over $\mathcal{S}$.
b_z^a	The belief transformation function, or the next belief state, given the currentinformation state b, action a and observation z.
d_n	A decision rule mapping each state to an action when there are n steps to go.
d_n^*	An optimal decision rule mapping each state to an action when there are n steps to go.
d_∞	A stationary policy employing the same decsion rule at every time step.

H	Entropy of a probability distribution. (Equation 6.5)
$\overline{H}$	Normalized Entropy of a probability distribution (Equation 6.8)
$\tilde{H}$	Scaled version of the normalized entropy of a probability distribution. (Equation 6.9)
I	The Kronecker delta or indicator function. (Equation 2.15)
$\mathcal{N}(\gamma)$	The set of all "neighbors" of a vector.
n	Index for iterations of dynamic programming.
O	Asymptotic big-oh notation.
O^t	The observation seen at time t.
$\mathcal{O}$	The observation function mapping action-state pairs into distributions over $\mathcal{Z}$.
$o(a, s, z)$	The observation probability of seeing observation z, when the action a resulted in a transition to state s.
$P^{a,z}$	A matrix of probabilities capturing the state transition and oberavtion probabilities for action a and observation z. (Equation 3.3)
$\Pr(X)$	The probability of event X.
$\mathbb{R}$	The real numbers.
$\mathcal{R}$	The immediate reward function.
$r(s, a)$	The expected immediate reward for performing action a in state s.
$R(\gamma, \Gamma)$	A subset of $\mathcal{B}$ where the vector γ dominates all other vectors in $\Gamma - \{\gamma\}$. (Equation 3.1)
$\mathcal{S}$	The set of states of a process.
S^t	The state of the system at time t.
SH	Expected state entropy. (Equation 6.6)
s, s', s''	A particular state in the set $\mathcal{S}$.
T	The horizon length in a finite horizon problem.
$\mathcal{T}$	The state transition function mapping state-action pairs into distributions over $\mathcal{S}$.
t	A time step or decision point.
$V_n(\pi, s)$	The n steps-to-go value of executing policy π starting in state s.
$V(\pi, s)$	The infinite horizon value of executing policy π starting in state s.
$V_n^*(s)$	The optimal n steps to go value of executing an optimal policy starting in state s.

$V^*(s)$	The optimal infinite horizon value of executing an optimal policy starting in state s.
$\widetilde{V}(s)$	An approximation to the optimal value function.
w	A function coverting a distribution over states into a distribution over actions. (Equation 6.4)
$\mathcal{Z}$	The set of observations in a process.
z, z'	A particular observation in the set $\mathcal{Z}$
Γ	A set of $\|\mathcal{S}\|$-vectors representing a set of hyperplanes of dimension $\|\mathcal{S}\|$.
γ	An $\|\mathcal{S}\|$-vector of real numbers representing a linear hyperplane of a value function.
η	The step-size or learning rate for incremental parameter adjustment methods.
Θ	Asymptotic big-theta notation.
κ	Entropy threshold in dual-mode control heuristics. (Page 265)
ν	A "neighbor" of a vector. (Definition 3.2.1)
Ξ	A decision process model.
$\Pi(\mathcal{A})$	A probability distribution over the action set.
$\Pi(\mathcal{S})$	A probability distribution over the state set.
$\Pi(\mathcal{Z})$	A probability distribution over the observation set.
π	A policy or sequence of decision rules for an MDP..
π^*	The optimal policy for an MDP.
π_{CO}	A decision rule for a COMDP derived from a POMDP. (Page 259)
ρ	The discount factor.
$\sigma(b, a, z)$	The probability of getting observation z, given that the current belief state is b and action a is taken. (Equation 2.12)
$\tau(s, a, s')$	The transition probability of ending in state s', given the starting state s and action a was chosen.
$\chi(\cdot)$	A mapping from observations to a set of vectors representing a particular choice of vectors from the set. (Equation 2.24)
$\psi(b, a, b')$	The transition function on information states. The probability that the resulting information state is b', given the current state is b and action a is taken. (Equation 2.14)
$\omega(b, a)$	The expected immediate reward accrued when the information state is b and the action a is taken. (Equation 2.16)

Operators

$a \cdot b$	The vector dot product of vectors a and b.
$A \oplus B$	The cross-sum of two sets of vectors, which is all ways of adding vectors from A to vectors in B.
$\overset{L}{>}$, $\overset{L}{<}$	Lexicographic comparison of two vectors as defined in Definition 3.1.2.

Acronyms

AI	Artificial Intelligence. (Page 2)
AV	Action voting POMDP control heuristic. (Page 260)
CEC	Certainty equivalent controller. (Page 257)
COMDP	Completely observable Markov decision process. (Page 15)
CUMDP	Completely unobservable Markov decision process. (Page 267)
DM	Dual mode POMDP control heuristic. (Page 266)
DP	Dynamic programming. (Page 20)
e.f.t.	Extended finitely transient property of a stationary policy. (Definition D.0.3)
f.t.	Finitely transient property of a stationary policy. (Page 45)
GCS	Generalized cross-sum, used in GIP. (Page 88)
GIP	Generalized incremental pruning. (page 84)
IP	Incremental pruning. (Page 81)
IP-LL	Incremental pruning where the two largest sets are always chosen. (Page 128)
IP-NCS	Incrmental pruning using NCS cross-sums. (Page 131)
IP-RR	Restricted region incrmental pruning. (Page 132)
IP-SL	Incremental pruning where the smallest and largest sets are always chosen. (Page 128)
IP-SS	Incremental pruning where the two smallest sets are always chosen. (Page 128)
LIN-Q	Linear Q-functions; a reinforcement learning algorithm for partially observable Markov decision processes. (Page 225)
LP	Linear programming. (Page 55)
MDP	Markov decision process. (Page 15)
MLS	Most likely state POMDP control heuristic. (Page 259)
NCS	Normal cross-sum, used in IP. (Page 119)
NDP	Neuro-dynamic programming. (Page 190)
OR	Operations research. (Page 1)
PI	Policy Iteration. (Page 26)
POMDP	Partially observable Markov decision process. (Page 29)
p.w.l.	Piecewise linear.

PWLC	Piecewise linear and convex. (Page 41)
Q-MDP	Control heuristic for a POMDP using the optimal Q-functions for the underlying completely observable MDP. (Page 261)
RL	Reinforcement learning. (Page 190)
RR	Restricted region cross-sum. (Page 120)
SPOVA	Smooth partially observable value approximation. (Page 227)
VI	Value Iteration. (Page 23)
WE	Weighted entropy POMDP control heuristic. (Page 268)

Index