8-6-2005

Algorithms for stochastic finite memory control of partially observable systems

Gaurav Marwah

ALGORITHMS FOR STOCHASTIC FINITE MEMORY CONTROL OF PARTIALLY OBSERVABLE SYSTEMS

By

Gaurav Marwah

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

August 2005

ALGORITHMS FOR STOCHASTIC FINITE MEMORY CONTROL OF PARTIALLY OBSERVABLE SYSTEMS

By

Gaurav Marwah

Approved:

Name: Gaurav Marwah

Date of Degree: August 08, 2005

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Eric A. Hansen

Title of Study: ALGORITHMS FOR STOCHASTIC FINITE MEMORY CONTROL OF PARTIALLY OBSERVABLE SYSTEMS

Pages in Study: 46

Candidate for Degree of Master of Science

A partially observable Markov decision process (POMDP) is a mathematical framework for planning and control problems in which actions have stochastic effects and observations provide uncertain state information. It is widely used for research in decision-theoretic planning and reinforcement learning. To cope with partial observability, a policy (or plan) must use memory, and previous work has shown that a finite-state controller provides a good policy representation. This thesis considers a previously-developed bounded policy iteration algorithm for POMDPs that finds policies that take the form of stochastic finite-state controllers. Two new improvements of this algorithm are developed. First improvement provides a simplification of the basic linear program, which is used to find improved controllers. This results in a considerable speed-up in efficiency of the original algorithm. Secondly, a branch and bound algorithm for adding the best possible node to the controller is presented, which provides an error bound and a test for global optimality.

Experimental results show that these enhancements significantly improve the algorithm’s performance.

DEDICATION

To my mother.

ACKNOWLEDGMENTS

I am grateful to Dr. Eric Hansen for his continuous support and help during the course of this thesis. It has been a privilege working with him, and a good learning experience both technically and morally.

I am also thankful to Dr. Lois Boggess and Dr. Gene Boggesss for their kind support all along.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

I. INTRODUCTION 1

II. BACKGROUND 4

2.1 Overview 4
2.2 Markov Decision Processes 4
2.2.1 Model 4
2.2.2 Complexity 6
2.2.3 Performance criteria 6
2.2.4 Policy Representation 8
2.2.5 Policy Evaluation 10
2.3 Related Work 11
2.3.1 Hansen's Policy Iteration Algorithm 11
2.3.1.1 Policy Evaluation 12
2.3.1.2 Policy Improvement 12
2.3.2 Stochastic Finite State Controllers (SFSC) 16
2.3.3 Platzman's algorithm 17
2.3.3.1 Platzman's Linear Program 17
2.3.3.2 De-randomization technique 20
2.3.4 Bounded Policy Iteration 23
2.3.4.1 Linear program for removing jointly dominated nodes . . 24
2.3.4.2 Poupart and Boutlier's improved linear program 24

III. IMPROVEMENTS TO BOUNDED POLICY ITERATION 26

CHAPTER Page

3.1 Overview 26
3.2 Simplified Linear Program 26
3.3 Branch and Bound algorithm 29
3.3.1 Preprocessing step 31
3.3.1.1 Building the search tree 32
3.3.1.2 Action-Observation pruning 32
3.3.1.3 Heuristic function 33
3.3.2 Lower bound 33
3.3.3 Check for dominance at decision nodes 34
3.3.4 Variations of branch and bound algorithm 35
3.4 Improved bounded policy iteration 35
3.4.1 Removing dominated nodes 36
3.4.1.1 Removing pointwise dominated nodes 36
3.4.1.2 Removing jointly dominated nodes 37
3.4.2 Improvement with constant memory 37
3.4.2.1 Improvement by de-randomization 37
3.4.2.2 Improvement by randomization 37
3.4.3 Improvement by adding memory 38
3.4.3.1 Platzman escape technique 38
3.4.3.2 Global search using branch and bound algorithm 38

IV. RESULTS 40

4.1 Overview 40
4.2 Reduction in the number of variables 41
4.3 Pruning using branch and bound 41
4.4 Timing results 42

V. CONCLUSION AND FUTURE WORK 44

REFERENCES 45

LIST OF TABLES

TABLE Page

2.1 Linear program to test for dominance 13

2.2 Hansen's Policy Iteration Algorithm 14

2.3 Algorithm for calculating Bellman residual 15

2.4 Platzman's linear program. 18

2.5 Platzman's linear program for each memory state. 19

2.6 Poupart's efficient linear program 25

3.1 Linear program to test for dominance of action-observation value vectors 28

3.2 Stochastic Policy Iteration 39

4.1 Benchmark problems 40

4.2 Reduction in the number of variables 41

4.3 Pruning using branch and bound 42

4.4 Timing results for the algorithm (in CPU seconds) 43

LIST OF FIGURES

FIGURE Page

1.1 Feedback required by the MDP to model the effect of agent actions 2

2.1 Policy Improvement by de-randomization 22

CHAPTER I

INTRODUCTION

The term planning refers to the formulation of a sequence of actions that fulfills a well-defined objective. Planning problems have traditionally generated lots of interest for researchers in the field of artificial intelligence (AI).

Complex planning problems involve actions that have stochastic effects, and planning in general involves uncertainties at several levels. This has led AI researchers to adopt algorithms from the field of decision-theoretic planning that make use of probability theory to model the uncertainties. The uncertainties require the models to have some mechanism for achieving feedback about the effect of actions taken by the agent. Markov decision processes (MDPs) provide a framework to model a feedback controller for decision-theoretic planning problems. Figure 1.1 provides an intuition about the role of MDPs in modeling planning problems.

There is an important class of MDPs where actions performed by the agent have stochastic effects, but there is no uncertainty in the information provided by the sensors. In other words, the agent has perfect knowledge about the current environmental state it is in. This class of MDPs is referred to as a completely observable Markov decision processes (COMDPs). There are various problem domains where this assumption is true, but in

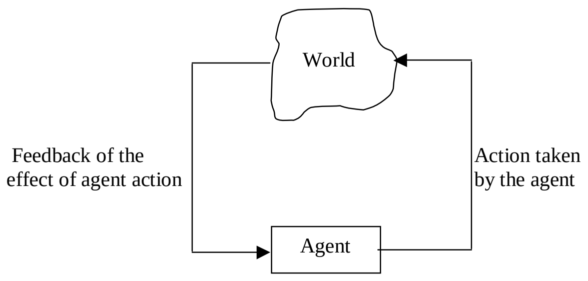
World
Feedback of the effect of agent action
Action taken by the agent
Agent

Figure 1.1 Feedback required by the MDP to model the effect of agent actions

general the sensors might not provide accurate information about the current environmental state. Another class of MDPs called partially observable Markov decision processes (POMDPs) serve as a useful framework for modeling these more difficult planning problems.

POMDPs assume that the agent's actions have stochastic effects as in COMDPs, and in addition, that the true state of the environment is hidden from the agent. This is because the sensors provide uncertain state information. This provides a more generalized framework for planning that encompasses both classical planning and COMDP. However, this drastically increases the computational complexity of the problem. This is because of the added task of information gathering that the agent has to perform. Most exact algorithms for solving POMDP problem instances can do so only for relatively simple problems.

The POMDP model is applicable to a large number of real world problems. Problems such as pursuit evasion [6], dialog management [13], robot navigation [14] and medical diagnosis are examples of problems with high practical significance which have made use of the POMDP model. However, most of these problems are very hard to solve exactly and therefore provide motivation for approximate solutions to POMDPs.

In this thesis, we will present two new techniques that improve the performance of the current state-of-the-art approximate algorithm for solving POMDP problems. The first technique reduces the size of the central linear program used by this algorithm. This results in huge speed up in efficiency. The second technique helps the original algorithm to break out of the local optima. Given enough resources, this enhancement essentially converts the approximate algorithm into an exact algorithm.

The rest of this document is organized as follows; Chapter 2 provides background information about MDP and describes the use of stochastic finite state controllers (SFSCs) as a method for policy representation and associated algorithms. This is followed by chapter 3, which describes some new techniques to make policy improvement using SFSC more efficient. Chapter 4 presents and discusses the results obtained. This is followed by the conclusion and future work.

CHAPTER II

BACKGROUND

2.1 Overview

This chapter presents some background information about Markov decision processes (MDPs). As previously mentioned, MDPs provide a useful framework for decision theoretic planning. This chapter starts with a brief description of the basic POMDP model. It then discusses various ways to represent policies (or plans) for POMDP problems. In addition, various algorithms for solving POMDPs are discussed. The chapter concludes with a brief discussion on the complexity of the POMDP problem and the algorithms used to solve it.

2.2 Markov Decision Processes

2.2.1 Model

MDPs provide a mathematical framework for planning and control problems involving agents whose actions have stochastic effects. If, in addition, the observations available to the agent provide uncertain state information, then the problem is categorized as a POMDP problem; otherwise it is categorized as a completely observable MDP (COMDP). The MDP model is widely used for research in decision-theoretic planning and reinforcement learning. This work concentrates on the POMDP model for planning.

A POMDP [16] is formally defined as a six-tuple (S, O, A, B, T, R) where:

- S is a finite set of possible states.
- O is a finite set of observations that provide incomplete information about the underlying states.
- A is a finite set of actions.
- T represents a state transition function that maps $S \times A$ into discrete probability distributions over S; let $Pr(s' \mid s, a)$ represents the probability that state $s' \in S$ is reached as a result of taking action $a \in A$ in state $s \in S$.
- R is the reward function that maps $S \times A$ into real numbers that represents the expected reward of taking an action $a \in A$ in a given state $s \in S$.
- B is an observation function that maps $S \times A$ into discrete probability distributions over O; let $Pr(o \mid s, a)$ represents the probability that observation $o \in O$ is observed when action $a \in A$ is taken and state $s \in S$ is obtained as a result.

It is important to notice here that a POMDP provides a more general framework than a COMDP. In fact, any COMDP problem can be modelled as a POMDP without the observation function B as defined above. Or, to be more precise, a COMDP can be modeled as a POMDP having the same set for observations O and states S and an observation function B with probability distribution as defined below:

$$Pr(o \mid s, a) = \left\{ \begin{array}{ll} 1 & if\ o = s; \\ 0 & if\ o \neq s. \end{array} \right\}$$

This simplification results in significant reduction in complexity for the COMDP model as compared to the POMDP model.

Since the true state is only partially observable in the POMDP framework, the agent uses the current observation together with the information of the previous history of actions and observations to obtain estimates of the possible current state. Based on this information it takes an action that maximizes its expected reward, which leads the agent into another hidden state.

2.2.2 Complexity

The general POMDP problem has been proven to be PSPACE-Complete for finite horizon POMDPs, and for infinite-horizon POMDPs, the problem is undecidable [7, 4].

2.2.3 Performance criteria

The reward function specifies the one-step consequences of an agent's actions. However, planning in general requires optimizing over multiple action choices for a substantial (possibly infinite) length of time. This is where the concept of performance criteria acquires significance. It provides a sense of the long term effect of taking an action in a given state. There are several choices for the performance criteria, which affects the complexity of the problem and also can potentially provide different policies. Before explaining the different performance criteria in detail, it will be useful to get more insight into the two aspects related to planning using the POMDP model as explained next.

First, because of the inherent uncertainty in the POMDP model, some way to represent the previous history of actions and observations is required. This will provide probabilistic information about the current state of the system (π_s will represent the probability of the state s being the current environment state). This aspect is discussed in detail in the section on policy representation. All these representations require memory in one form or another.

Secondly, performance criteria by definition require some way to look ahead to the possible expected cumulative reward of taking an action at the present instance. This is possible using probability theory over the transition, observation and reward functions. However, the extent of look ahead may differ for different performance criteria, some of which are described next.

Some of the most widely used performance criteria include expected cumulative rewards over a finite horizon, over an infinite horizon and over an indefinite horizon. All these are explained briefly below. Apart from these some other criteria such as average reward per time step are also possible.

Performance over a finite horizon involves a look ahead to a finite length of time steps, which provides substantial computational leverage. The expected cumulative reward of taking an action in the present state assuming a finite horizon is given by:

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

On the other hand, performance over an infinite horizon does not limit the look ahead to a finite number of time steps. Instead, it uses a discount factor to reduce the effect of actions at distant time steps. This generalization increases the complexity of the problem. The expected cumulative reward of taking an action assuming an infinite horizon and a discount factor β is given by:

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

The discount factor takes a value between 0 and 1 which ensures that at every time step the effect of an action decreases geometrically.

Expected cumulative reward over an indefinite horizon is closely related to that over an infinite horizon. It assumes the presence of stopping or absorbing states which can provide a limit to the amount of look ahead.

One important subtlety worth mentioning here is that for this work we will model the problem in discrete time steps. There are other methods that work with continuous time models.

2.2.4 Policy Representation

As there is uncertainty in the underlying state of a POMDP, a policy for a POMDP would need some way to estimate the underlying state of the system. One way to do this might be to store the entire history of actions and observations, and to use this in order to estimate the hidden state. Such policies are known as history policies [10]. The number of possible histories grows exponentially with the horizon and as such these policies become intractable for reasonably large horizon length.

One way to avoid the problems related to explicit storage of action-observation history is to use Bayesian updating of probability distributions over underlying states, called information states. The information state will be represented by π where $\pi(s)$ will represent the probability of s being the underlying state. Depending on the action taken in the current information state and the resulting observation, a simple probabilistic update will convert it into a new information state. The information state provides a sufficient statistic in the sense that the POMDP model can be represented as a Markov chain with the information states as the states of the chain. However, the information state space is continuous and therefore there are an infinite number of information states and corresponding states in the Markov chain.

Other schemes for policy representation use memory in some form or other to cope with partial observability. This is not the case with policies for COMDP, which can be represented by deterministic memoryless or reactive policies. The choice of action for such policies depends on the current state only. The reason behind the applicability of deterministic memoryless policies to COMDP is that they can be represented by Markov chains, where each state of the chain corresponds to an actual environment state, and therefore the choice of action depends on the current state only, which is completely observable in COMDPs.

However, because of the uncertainty associated with hidden states in a POMDP, deterministic memoryless policies are not very useful for POMDPs. Littman [8] and Singh [15] experimented with the use of memoryless stochastic policies for POMDP. Their approach induces randomization into a policy by randomly choosing among different actions on the same observation. It has been shown to perform better than a deterministic memoryless policy that always performs the same action on same observation. In addition, stochastic memoryless policies do not get trapped in deterministic loops, which means they can get out of a repetitive sequence of actions that does not lead towards the goal. However, in most cases a stochastic memoryless policy cannot perform better than an optimal deterministic policy given sufficient memory. Nevertheless, given the same amount of memory a stochastic policy performs at least as well or better than a deterministic policy and this forms the motivation for this work [8]. The reason behind this is that use of a stochastic controller adds randomization to a policy. In the case of complete absence of knowledge, randomization may be the best strategy to follow. In our case, where there is lack of knowledge (due to uncertainty), some amount of randomization may help.

One way to provide memory is to use a finite state controller to represent a policy for a POMDP problem. Several approaches for generating policies for a POMDP (such as Hansen's policy iteration [4]) make use of deterministic finite state controllers for representing memory. However, the complexity of these algorithms depends on the size of the controllers, which can become very large for moderately hard problems.

An intuitive approach to deal with limitations of memory may be to use a stochastic finite state controller for policy representation which may help in scaling up the algorithms for POMDPs [11]. This is the idea pursued in this thesis.

2.2.5 Policy Evaluation

Having gotten an insight into various criteria to judge the expected future cumulative effect of taking an action at the present time step, the next step is to describe ways to judge the complete policy as a whole. This aspect is covered under policy evaluation, and provides a mechanism to compare policies with a view to finding the optimal one.

Policy evaluation for a POMDP uses the concept of a value function. A value function can be obtained for each possible policy in the policy space; the objective is to find the optimal policy using these value functions. A value function for an arbitrary policy δ in the POMDP policy space provides a value for each possible information state.

For this work, we will deal with policy evaluation of a policy represented as a finite state controller. It is important to mention here that the memory states of the finite state controller together with the underlying states of the POMDP problem form a cross product MDP [4]. That is they form a Markov chain where each possible pair of a memory state and an underlying state represent a unique state of the chain. If N represents the set of memory states in the controller, then there will be $\mid N \mid\mid S \mid$ states in the Markov chain. A value will be defined for each of these states in the Markov chain. This can be represented by use of an $\mid S \mid$ dimensional vector corresponding to each of the memory states. More specific details are provided in the section on related work.

2.3 Related Work

2.3.1 Hansen's Policy Iteration Algorithm

Hansen [4, 5] developed a policy iteration algorithm that uses a deterministic finite state controllers to represent policies. In addition to computational benefits, this algorithm considerably simplifies the process of policy evaluation. The algorithm performs a dynamic programming update on the value function V^{δ}, representing the current controller δ, to obtain all possible vectors that can possibly improve the controller. Each of these vectors corresponds to a memory state that can be added to the controller. The algorithm iteratively improves the value function by improving the finite state controller in accordance to this newly obtained set of vectors. At each iteration, the previous controller is converted into an improved controller by utilizing three operations: merging, adding, and pruning of memory nodes from previous iteration. The algorithm is described in Table 2.2.

2.3.1.1 Policy Evaluation

Hansen's algorithm considerably simplifies the process of policy evaluation. Using this algorithm, policy evaluation is equivalent to solving the following system of linear equations, where i is the index of the memory state, $\alpha(i)$ is the action taken by the deterministic controller in memory state i and $\tau(i, o)$ gives the final memory state after the transition from memory state i on observing observation o.

$$\gamma_i(s) = r(s, \alpha(i)) + \beta \sum_{s' \in S, o \in O} Pr(s' \mid s, \alpha(i)) Pr(o \mid s', \alpha(i)) \gamma_{\tau(i,o)}(s') \tag{2.3}$$

The value function can then be defined for each possible information state and will be given by following equation, where the set Γ is the set of $\mid S \mid$ dimensional vectors corresponding to each memory state.

$$V(\pi) = \max_{\gamma \in \Gamma} \sum_{s \in S} \pi(s)\gamma(s) \tag{2.4}$$

2.3.1.2 Policy Improvement

As discussed in the section on policy evaluation, the value function can be represented by a finite number of vectors, each of which can be represented by a memory state of a finite state controller. The dynamic programming (DP) update involves converting a given set of these vectors into another set of vectors that improves the value function for at least some of the information states and does not decreases it for any of the information states. The DP update forms the basis of the iterative algorithms for solving POMDPs.

Variables: ϵ, $b_s, \forall\ s \in S$

Maximize: ϵ

Constraints: $\epsilon \leq \sum_{s \in S} b_s \cdot V^n(s) - \sum_{s \in S} b_s \cdot V^{'}(s), \forall n \in N$

$\sum_{s \in S} b_s = 1,$

$b_s \geq 0, \forall s \in S$

Table 2.1 Linear program to test for dominance

Hansen's policy iteration algorithm takes the DP update a step further. Instead of interpreting it as a method of adding nodes, it interprets the DP update as an improvement of the finite state controller by using three operations: changing, pruning and adding nodes. The algorithm uses incremental pruning [3] as the default method for generating a list of all possible nodes that can be added to the controller. Incremental pruning is widely accepted as the most efficient method in practice for DP update over a variety of problems. The algorithm will work with any method of performing complete or partial DP update.

The algorithm primarily has to make two decision choices at each iteration. Firstly, it has to decide which of the new nodes obtained as a result of DP update will replace the

1. Input : An initial finite state controller δ, a parameter ϵ providing bound on optimality of the policy represented by the controller.
2. Policy evaluation: Using equation 2.3 compute the value function V^δ representing controller δ.
3. Policy Improvement:
   (a) Perform dynamic update on V^δ to obtain a set of vectors Γ' representing a new value function V'.
   (b) For each vector γ' in Γ':
      i. If the action and successor links associated with it are same as those of any other memory state already in δ, then keep that memory state in δ'.
      ii. Else if the vector γ' pointwise dominates any other vector associated with a memory state in δ, then change the action and successor links associated with that memory state to those that correspond to γ'(If it pointwise dominates more than one memory state in δ then merge all those memory states).
      iii. Else add a single memory state to δ' that has action and successor links associated with γ'.
   (c) Prune any memory state of δ' for which there is no corresponding vector in Γ', only if it is not reachable from any other vector in Γ'.
4. Termination test: Calculate the Bellman residue according to 2.3, if it is less than or equal to $\epsilon(1-\beta)/\beta$, then go to step 5. Else set δ to δ'. If some node was changed in step 3b, go to step 2; otherwise go to step 3.
5. Output : An ϵ -optimal finite state controller.

Table 2.2 Hansen's Policy Iteration Algorithm

earlier nodes. For this, it utilizes the concept of pointwise dominance. A node pointwise dominates another memory node if its value function vector provides a higher value for all the underlying states. A node that pointwise dominates another node can replace that node without decreasing the value function of the controller. Most of the time, however, a new vector improves the value function for a part of the belief region only. In this case, it is simply added to the controller without replacing any other node.

The second decision that the algorithm has to make is the point of termination of the algorithm. For this purpose, the algorithm makes use of the Bellman residual (see Table 2.3) to decide on ϵ optimality [16, 4]. If the best improvement for all of the nodes in the controller is less than $\epsilon(1-\beta)/\beta$, then the policy represented by the controller is guaranteed to be ϵ-optimal.

One bottleneck of this algorithm is that the complexity of the algorithm depends upon the number of nodes in the controller, which can become very large or possibly infinite. Therefore, some way to reduce the number of memory states could be useful. The algorithms described in the next section can potentially achieve this objective.

```
1. Input : A set of vectors $\Gamma^n$ representing value function $V^n$ and a set of vectors $\Gamma^{n-1}$
   representing value function $V^{n-1}$.
2. residual := 0
3. For each $\gamma$ in $\Gamma^n$
   (a) Solve the linear program of Table 2.1 with inputs $\gamma$ and $\Gamma^{n-1}$.
   (b) If ( d $>$ residual) then residual := d
4. Output : residual
```

Table 2.3 Algorithm for calculating Bellman residual

2.3.2 Stochastic Finite State Controllers (SFSC)

There is some evidence in theory which suggests that, given the same amount of memory, a stochastic finite state controller can perform better than a deterministic finite state controller [8, 11]. Therefore, use of a stochastic finite state controller may help in scaling up algorithms for POMDPs.

This section will discuss some of the algorithms that find stochastic policies for POMDPs. Not much work has been done to obtain randomized policies in the form of stochastic finite state controllers. The two approaches in the literature are attributed to Meuleau's [9] and Baxter's [1] gradient based approach that does not require a model, and Platzman's [11] and Poupart's [12] linear programming based approach which does. Since we are considering planning with a model, we will concentrate on the latter approach.

Before going into details of the algorithm it is worth mentioning the difference in policy representation of a determinstic controller vs. a stochastic controller. Each node in a deterministic controller performs an action $a \in A$, and based on observation $z \in Z$, it makes a deterministic transition to a node $n \in N$. The stochastic controller consists of stochastic nodes which perform a probabilistic action based on distribution over a set of possible actions $|A|$. For each observation, the node transitions are also governed by probability distribution over set of nodes N.

Comparing the two methods of policy representation, it can be seen that any stochastic node can be represented as a convex combination of all the possible deterministic nodes that can be present in the controller. This number equals $|A|\ |N|^{|Z|}$

If a stochastic finite state controller is used instead of a deterministic one then policy evaluation requires solving a set of equations, where each equation is given by:

$$\gamma_i(s) = \sum_{a \in A} \alpha(a,i) r(s,a) + \beta \sum_{a \in A} \alpha(a,i) \sum_{s' \in S, z \in Z} (Pr(s' \mid s,a) Pr(z \mid s',a) \sum_{j \in N} \tau(i,z,j) \gamma_j(s'))) \tag{2.5}$$

Here, $\alpha(a,i)$ represents the action probability and is the probability of taking action a in memory state i, and $\tau(i,z,j)$ is the node transition probability which is the probability of making a transition to memory state j on observing observation z in memory state i.

2.3.3 Platzman's algorithm

Platzman devised a linear programming based algorithm that provides a sub-optimal stochastic policy for POMDP [11]. The algorithm uses a linear program to improve each memory node of the controller, until it gets into a local optimum. At this point it adds a node to the controller using an escape technique based on what we call de-randomization, described latter. If this fails to find an improvement, then this algorithm uses systematic enumeration to break out of the local optimum, which is not very attractive. The algorithm also provides an error bound on performance.

2.3.3.1 Platzman's Linear Program

Table 2.4 provides the mathematical formulation of Platzman's linear program. The variable ϕ_j^n represents the probability of occurrence of event $j \in J$ in memory state n. A

Variables: ϵ, ϕ_j^n, $\forall\, n \in N$ and $j \in J$.

Maximize: ϵ

Constraints: $\epsilon \leq \sum_{j \in J} \phi_j^n V_{s,j}^n,\ \forall s \in S, n \in N$

$\sum_{j \in J} \phi_j^n = 1,\ \forall n \in N$

$\phi_j^n \geq 0,\ \forall j \in J, n \in N$

Table 2.4 Platzman's linear program.

member j of the event set J for a memory state signifies the choice of an immediate action $a \in A$ followed by the next memory state $n \in N$ for each of the observations $z \in Z$. This representation combines the action and node transition probabilities into an event. The cardinality of the event set is $\mid A \mid\mid N \mid^{|Z|}$. The linear program tries to optimize over all the elements of the event set for each memory state. Platzman also describes a way to divide this linear program into a separate, smaller linear program for each memory state which considerably simplifies the complexity of the linear program. This simplified linear program is given in Table 2.5, which needs to be solved for each memory state.

The linear program tries to maximize the value of variable ϵ that represents, for a given memory state, the minimum improvement possible in the value for any of the underlying states. $V_{s,j}$ represents the maximum improvement in state s that is possible if event j is chosen deterministically in the given memory node. The variable ϕ_j represents the probability of taking event j in the current memory state. The product $\phi_j V_{s,j}$, therefore represents the expected improvement for state s if the event j is chosen with a probability ϕ_j from the set of events J.

Variables: ϵ, $\phi_j, \forall\, j \in J$.

Maximize: ϵ

Constraints: $\epsilon \leq \sum_{j \in J} \phi_j V_{s,j},\ \forall s \in S$

$\sum_{j \in J} \phi_j = 1$

$\phi_j \geq 0,\ \forall j \in J$

Table 2.5 Platzman's linear program for each memory state.

A closer look at the constraints in Platzman's linear program reveals that it adjusts these probability parameters in a way that maximizes the improvement in the value function for any one or more of the underlying states without decreasing it for any other. If the program is successful, then it finds a new set of probability values for the event parameters of the current memory node. In that case, the resulting value function vector for the current memory node will pointwise dominate the previous vector for the current node. The constraint for pointwise dominance, however, puts a strong restriction on the linear program, which may affect its ability to find an improved vector.

Another point worth mentioning here is that Platzman's program follows an iterative approach to improve the controller. At each iteration, the linear program causes a local improvement to the value function. Therefore, the use of Platzman's algorithm for obtaining the best stochastic controller will require executing the linear program given in Table 2.5 $\mid N \mid$ number of times at each iteration. The number of iterations cannot be known in advance.

Each event j corresponds to a deterministic node that possibly could be added to the controller as a result of a DP update. Since the linear program tries to find an optimal probability distribution over the event set, this means that the linear program is trying to find a convex combination of nodes that would be generated as a result of DP back up without actually performing the DP update.

Platzman's linear program has $|S| + 1$ constraints and $|A|\ |N|^{|Z|} + 1$ variables. This clearly makes the program intractable for a moderately large number of memory states, and some way to simplify this linear program is needed.

2.3.3.2 De-randomization technique

The discussion in this section, at first, might appear contrary to the fact stated in the previous section that randomization in general can improve the controller without adding memory. Although this is true, it does not mean that ”any” stochastic controller will have that property. What it means is that an ”optimal” stochastic controller would be no worse than an optimal deterministic controller of the same size. This becomes interesting in view of the fact that stochastic controllers obtained as a result of linear programs explained in the previous sections are locally optimal. This locally optimal controller might have some redundant randomization that could be removed which can possibly improve the controller.

An important realization that helps to understand the point made above is that the linear program described in the previous section imposes tough constraints on improvement. It tries to improve a node by finding a convex combination that would guarantee improvement for each of the belief states. These tough constraints guarantee that the value function of the controller will not decrease at any iteration. However, it seems that such constraints may sometimes cause the linear program to fail to find an improvement, when an improvement in the overall value function is in fact possible.

Recall that each node of the controller corresponds to an $|S|$-dimensional vector. The controller as a whole represents a value function obtained as a result of these vectors. This value function is piecewise linear and convex. When this is viewed along with the linear programs to improve the stochastic nodes, it implies that we do not have to improve a stochastic node for each possible belief state. Instead, if we can improve it in a belief region where it already dominates, that will give a guaranteed overall improvement. This will also relax the constraints on the linear program, thereby increasing the chances of finding an improvement.

Based on the reasoning given above, an alternative way of improving the controller is also possible and is described next. Figure 2.1 helps in illustrating this point. It represents the value function for a two state POMDP problem. There are three nodes in the controller represented by vectors $V1$, $V2$ and $V3$. These vectors are convex combinations of vectors $V1'$, $V2'$, $V3'$ and $V4'$ which represent the nodes obtainable after DP update that would improve the controller for some belief region. Notice that the current controller (shown in leftmost diagram) has achieved a local optimum here and no convex combination can further improve any of the nodes for all the belief states.

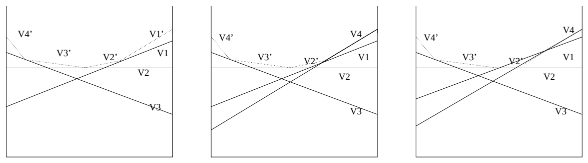

Figure 2.1 Policy Improvement by de-randomization

Now consider the situation shown in the middle diagram in Figure 2.1. The only difference in this controller is that the DP updatable node $V1'$ from the earlier controller is one of the nodes in the present controller and is represented as $V4$. $V2'$, $V3'$ and $V4'$ are the three nodes possible after DP update. Notice that the controller is still in a local optimum. None of the nodes can be improved by using a linear program that tries to improve it for all possible belief states.

However, notice that vector $V1$ is a convex combination of $V4$ and $V2'$. Since vector $V4$ is already present in the current controller, we can still obtain guaranteed improvement by improving $V1$ for all the belief regions, except those where $V4$ dominates $V1$. One simple way of achieving this might be to remove the deterministic component corresponding to vector $V4$ from the probability distribution of vector $V1$ and normalizing it over the remaining vectors involved in the convex combination.

The rightmost diagram in Figure 2.1 represents vector $V1$ after normalization. Notice that $V1$ is now same as vector $V2'$. Also, since $V2$ is now a convex combination of $V1$ and

$V3'$, it can also be improved along the same lines. The same could be done for $V3$ as well at the next step.

The above technique takes into account the piecewise linear and convex property of the value function and tries to improve the upper surface of this function. However, there is still one issue that needs some clarification. Consider the middle diagram in Figure 2.1 again. Notice that although we have achieved overall improvement, the value of vector $V1$ has decreased for some of the belief regions. Any vector in the controller that has a component corresponding to vector $V1$ in its probability distribution can still possibly lose value in those belief regions. This problem can however be easily overcome by following this technique with the linear program described in previous section. In such a scenario, we would be able to find an improved probability distribution for all the negatively affected vectors. The new probability distribution will have a component corresponding to vector $V4$ in addition to others that were previously there.

Lastly, it was assumed that the controller had a deterministic node like $V4$ on which the derandomization technique is based. However, if such a node is not present then we can always add it to the controller and the rest of the procedure remains same.

2.3.4 Bounded Policy Iteration

Recently, Poupart and Boutlier [12] proposed a Bounded Policy Iteration algorithm that uses linear programs along the same lines as Platzman. There are three important contributions of this work. Firstly they provide a simple linear program that could be used to remove jointly dominated nodes from the controller. Secondly they provide a much more efficient linear program then Platzman's linear program for improving nodes for all belief states. Thirdly they provide an escape technique to break out of local optima.

2.3.4.1 Linear program for removing jointly dominated nodes

This linear program is the dual of the linear program to test for dominance given in Table 2.1. The original linear program checks each node to test if it dominates in any belief region. The dual will test if the node is dominated by other nodes in the controller, and if so the solution of the dual will provide a convex combination that gives maximum improvement. We can replace every dominated node by this convex combination for all node transitions, without losing the quality of the policy.

2.3.4.2 Poupart and Boutlier's improved linear program

Poupart and Boutlier provide two linear programs for improving a node in the controller. The naive linear program proposed by them is the same as that given in Table 2.5. They also provide an efficient linear program to improve a node in the controller. The modified linear program reduces the number of variables significantly from $|A|\ |N|^{|Z|}$ to $|A|\ |N|\ |Z|$.

Table 2.6 provides details of this linear program. The new linear program achieves efficiency by segregating and rearranging the components of the event set J of Platzman's algorithm. The main idea is that every event in Platzman's algorithm can be represented as a sum of $|A||N|Z|$ different vectors. Therefore, we can rearrange them to dramatically reduce the number of variables.

Variables:ϵ, ϕ_{a,n_z}, $\forall$, $a \in A$, $n \in N$, $z \in Z$

Maximize: ϵ

Constraints: $\epsilon \leq \sum_{a \in A, n \in N, z \in Z} \phi_{a,n_z} \cdot V_{s,a,n_z}, \forall s \in S$

$\sum_{a \in A, n \in N, z \in Z} \phi_{a,n_{z)}} = 1$

$\phi_{a,n_z} \geq 0, \forall a \in A, n \in N, and, z \in Z$

Table 2.6 Poupart's efficient linear program

The variable ϕ_{a,n_z} represents the probability of taking action a and making a transition to node n on observing z.

CHAPTER III

IMPROVEMENTS TO BOUNDED POLICY ITERATION

3.1 Overview

This chapter presents some enhancements of the bounded policy iteration algorithm (BPI). The first enhancement concerns the linear program given in Table 2.6, which is the single most time consuming step in BPI. A simplification of the linear program is presented that results in a considerable speed-up in efficiency. The second enhancement concerns the method used to break out of the local optimum by adding a node to the controller. A branch and bound algorithm is presented that adds the best possible node to the controller, and also provides an error bound and a test for global optimality.

3.2 Simplified Linear Program

This section presents a technique that reduces the number of variables in the linear program given in Table 2.6. Recall that this linear program finds, for each node, an improved probability distribution over the nodes created by the DP update, without performing the DP update as mentioned in the section on bounded policy iteration in Chapter 2. The linear program considers all possible transitions for each action and observation pair. That is why the number of variables in the linear program is $|A||N||Z|$. In other words, there are $|N|$ variables for each action-observation pair. However, the number of variables for each action-observation pair that will have non-zero values in the improved distribution is bounded by the number of underlying states. This is because if there are more than $|S|$ non-zero variables for any action-observation pair then the number of analogous non-zero variables in Platzman's linear program would be more than $|S|$. But this is not possible because the number of non-zero variables is bounded by the minimum of the number of variables or the number of constraints in the linear program. In Platzman's linear program, the number of constraints is equal to the number of states. Therefore, another way to look at this is that the degree of randomization is limited by the number of states. For example, for a controller with 100 nodes, 3 actions, 2 observations and 4 states, the number of variables in Poupart's linear program would be 600. However, the number of non-zero variables would be at most 24 (and usually is much less than that). This provides the main motivation behind this simplification. It is worth clarifying that the above example does not suggest that the number of variables in the linear program is bounded by the maximum number of non-zero variables. In this example, the number of actual variables could be more than 24. In the worst case, it could be 600. However, the maximum number of non-zero variables is not a lower bound on the number of variables either. It is possible to reduce the number of variables below 24 as well.

The main idea behind this technique is that the test for dominance presented in Chapter 2 (see Table 2.1) can be extended to each action-observation pair. In other words, for each action-observation pair, it is possible that the partial value functions corresponding to several variables could be dominated by other variables for that action-observation. There is no need to include such variables in Poupart's linear program.

Theorem 1 *The removal of variables corresponding to dominated partial vectors for each action-observation will not reduce the maximum improvement found by the linear program.*

Proof: It can be shown that none of the variables removed in such a way will have non-zero value in the solution of the original linear program. If this were not true, than we could always replace such variables with a convex combination that dominates it, and find a better improvement. Since the linear program has already found the best improvement this is contradictory. Therefore, it may be deduced that removed variables will always have zero values in the solution of the linear program. Therefore, their removal will not make any difference to the quality of the solution. Q.E.D.

The linear program to test for dominance of value vectors for a particular action-observation is given in Table 3.1.

Variables: ϵ, $b_s, \forall\ s \in S$
Maximize: ϵ
Constraints: $\epsilon \leq \sum_{s \in S} b_s \cdot V^m_{a,z}(s) - \sum_{s \in S} b_s \cdot V'_{a,z}(s), \forall m \in \delta$
$\sum_{s \in S} b_s = 1,$
$b_s \geq 0, \forall s \in S$

Table 3.1 Linear program to test for dominance of action-observation value vectors

In practice, a large number of variables can be pruned using this approach and it provides considerable speed up in solving the linear program. However, there are two potential drawbacks of this approach. The first one is that in the worst case none of the variables can be pruned. But experiments described in Chapter 4 show that usually there is a considerable reduction in the number of variables. A second potential drawback is that this technique requires solving additional linear programs (one per variable); in fact, it requires solving $|A||N||Z|$ linear programs. However, these linear programs are much smaller; the number of variables is equal to the number of underlying states, and thus they can be solved relatively quickly. Moreover, these linear programs need to be solved only once for all the nodes in the controller. In effect, the added number of linear programs is $|A||Z|$ per node. Chapter 4 provides empirical results that show the effectiveness of this technique.

3.3 Branch and Bound algorithm

As described in Chapter 2, the linear program for pointwise improving each node of the controller, while keeping other nodes fixed, can get the controller into a local optimum. There are several ways to break out of the local optimum, such as Platzman's de-randomization technique and Poupart's single-step look ahead search for improvement at tangent belief states. However, none of these escape techniques guarantees improvement, when in fact the policy can be improved. This problem particularly becomes interesting when all the nodes in the locally optimal controller are deterministic. This point gains more importance in view of the fact that a truly stochastic controller can never be optimal. A truly stochastic controller is one in which there is at least one node which is represented as a convex combination of more than one deterministic node. Therefore it is reasonable to conclude that the controllers tend to become deterministic as they reach near-optimality. However, at this moment this idea requires further thought and is left as a future work.

Theorem 2 *A truly stochastic finite state controller can never be optimal provided there are no dominated nodes in the controller.*

Proof: Let us assume that such a controller is optimal. Then, by definition there is at least one node in the controller, which is represented as a convex combination of more than one deterministic node. Therefore, for every belief point, this node would be dominated by at least one of the deterministic nodes in the convex combination. Therefore, adding all such deterministic nodes will result in overall improvement of the policy, provided that the stochastic node itself dominates for some belief region as stated in the theorem. Q.E.D.

Another limitation is that these methods do not provide an error bound or a test of convergence to global optimality. A full DP-update provides an error bound and test of convergence to optimality by finding all the deterministic nodes that can be added to the current controller. However, in our case, we are interested in finding the best single node which will help us to break out of the local optimum and also provide an error bound and convergence test. In fact, considering that the global search is a computationally expensive step, we can terminate the search as soon as we find the first node that improves the controller for some belief state. This will also provide a loose upper bound on optimality.

A naive way of doing global search is systematic enumeration of all the deterministic nodes that can be added to the controller. This makes it possible to compute an error bound and test for global optimality in the course of searching for the single best node. The depth of the search tree is $1 + |Z|$ since we need to make a decision about the action to be taken and a decision about node transition for each observation. The branching factor at depth one would be $|A|$ and for the rest of the tree the branching factor would be $|N|$, since for each observation we can make a transition to any of the $|N|$ nodes. The number of leaf nodes in this tree is $|A||N|^{|Z|}$.

The same tree can be searched more efficiently using depth first branch and bound to prune branches of the tree that cannot lead to an improved node. The branch and bound algorithm is described in the following sections. Its time complexity depends on the number of nodes visited, which in the worst case is equal to the number of leaf nodes as given above. However, a lot of pruning is usually possible. The details of the algorithm are described next.

3.3.1 Preprocessing step

The preprocessing step creates the search tree, performs action-observation pruning, and calculates the heuristic function. These preprocessing steps are described below.

3.3.1.1 Building the search tree

The search tree could be thought of as a tree with $|A||Z|$ decision nodes, corresponding to each action-observation pair. At each decision node, we have to decide the choice of memory node in the current controller that the new node would transit to for the action-observation corresponding to that decision node. In the most naive form, this step would have to maintain a list of all the current memory nodes for choice at each decision node. However, we combine this step with action-observation pruning as described next which makes the search much more efficient. Note that the size of the search tree is relatively small.

3.3.1.2 Action-Observation pruning

As mentioned before, for each observation there are $|\delta|$ branches emerging from the corresponding decision node in the search tree. However as explained in the section 3.2, for each action-observation pair a lot of branches, whose corresponding partial value vectors are dominated, could be pruned. We could use the same linear program used to reduce the number of variables in (Table 3.1), to perform this pruning. In fact we do not have to perform this step again as it was performed for reducing the number of variables. The linear programs would have to be performed again, if there is any change in the controller. Since in our algorithm provided in 3.2 the branch and bound algorithm would be used only when all other methods have failed to find an improvement, we can assume that there is no change in the controller. Another point to mention is that the amount of pruning obtained would be exactly same as the pruning in the number of variables defined in section 3.2.

3.3.1.3 Heuristic function

The heuristic function would provide an upper bound on the improvement that could be obtained by following the rest of the search path from a particular decision node. This upper bound will help in further pruning the search tree. We use an S dimensional heuristic. For each action-observation pair, we assign the value function of the best node that can be added for each environmental state s, to the s^{th} dimension in the heuristic function for that action-observation. The heuristic function at any decision node would be the sum of the heuristics for each observation yet to be expanded. The heuristic function defined in such a manner will make sure that no matter what the optimal node is, its value function for any action-observation could not be better than the heuristic for any state. The reason behind this is that none of the partial value vectors for any action-observation can pointwise dominate a convex combination of nodes with maximum values at underlying environmental states.

3.3.2 Lower bound

In addition to action-observation pruning, a second level of pruning could be obtained by using a lower bound on improvement. As soon as we get the first node that finds an improvement for some belief state, we can use its improvement as a lower bound to prune the search tree. The lower bound will increase every time we find a better node. In fact, given that everything else remains the same, much more pruning could be obtained if we expand the best nodes early in the search. There might be some ways possible that could be considered in future work.

3.3.3 Check for dominance at decision nodes

A third level of pruning could be obtained at each decision node. The key point here is that we can prune a branch of the search tree if the maximum possible improvement from that branch is less than the current lower bound. The test to decide whether to explore a branch in the search tree any further will use this principle along with the heuristic defined above. This test is exactly same as the linear program to test for dominance of a memory node 2.1. However, since we have not generated the deterministic node completely we do not know the true value function to be tested. Recall that the value function for each node is an $|S|$ dimensional vector, with each vector element representing the value of being in that node in that state. At any decision node in the search tree, we have a partial value function for the decision nodes preceding that node. In addition, an upper bound on the partial value function for the rest of the decision nodes in that branch could be obtained by summing their heuristic for each state. The sum of the partial value function for the expanded decision nodes and the upper bound for the unexpanded nodes will give an upper bound for the value for all the states. We will then create an imaginary node with this relaxed value function and test it for dominance in the current controller. If the imaginary node is dominated, then all the leaf nodes in that branch will be dominated and therefore we can prune it.

3.3.4 Variations of branch and bound algorithm

There are several variations possible to the basic branch and bound algorithm described above. One such variation is that we can perform a full DP update instead of adding only the best node. Approaches such as incremental punning usually perform very well for doing a full DP update. One interesting comparison would be to compare the branch and bound with incremental pruning when we want to add only the best node. This is left as future work. Another variation is to return with the first node that finds an improvement over the current controller. This could potentially provide a great deal of speed-up especially when the number of such nodes is very large. The improvement found by this node will still provide an upper bound on the Bellman residual, and therefore will provide a bound on the quality of the controller.

3.4 Improved bounded policy iteration

In this section we present a generalized bounded policy iteration algorithm. Several variations of this basic algorithm are possible, and many of them are explained. The algorithm is given in Table 3.2. The basic idea of the algorithm is to find an improvement using the least computationally expensive method. However, such a method when used alone can get stuck in a local optimum. In such a case the next more expensive method is tried. If everything else fails a global branch and bound algorithm would be used which will provide global improvement if it is possible. Since the complexity of the algorithm depends upon the number of nodes, we want to keep it to as low as possible. Therefore, the algorithm tries to improve the controller as much as possible for a given number of memory states before adding new memory states. The algorithm utilizes three methods to improve the controller as described next, in increasing order of computational complexity.

3.4.1 Removing dominated nodes

This step removes any unnecessary nodes from the controller. The algorithm uses two methods to do that: an efficient method (as given in Table 2.2 to remove pointwise dominated nodes and a method that requires solving a linear program but removes jointly dominated nodes (see Table 2.6). Note that the latter is a more general case.

3.4.1.1 Removing pointwise dominated nodes

This was described in the last chapter in the section on Hansen's policy iteration algorithm. For each memory state m in δ, this technique checks to see if the vector associated with memory state m pointwise dominates any other vector associated with a memory state in δ, and if so then change the action and node transition probabilities associated with that memory state to those that correspond to m. (If it pointwise dominates more than one memory state in δ then merge all those memory states).

3.4.1.2 Removing jointly dominated nodes

This method is described in Section 2.3.4.2 for removing jointly dominated nodes. This method solves the linear program in Table 2.6 for each memory state m in δ; and if the vector associated with m is dominated by any convex combination of other memory states in δ, then it replaces the node transition probabilities for all the transitions that point to m with the probabilities obtained in the convex combination.

3.4.2 Improvement with constant memory

3.4.2.1 Improvement by de-randomization

This method is described in Section 2.3.3.2 . This method tries to find whether there are some deterministic memory nodes in the controller, and there are memory states in δ that have probabilistic components to those deterministic modes. If it finds such a case, then we can remove such probabilities and normalize the difference over the rest of the components.

3.4.2.2 Improvement by randomization

If all the previous methods fail then, for each memory state in δ, we solve the linear program from Table 2.6 with set of variables obtained from Table 3.1 to improve the value function associated with it for each state. If we find an improvement then we change the action and node transition probabilities associated with that memory state to those that correspond to improved values. This will guarantee improvement without making the controller worse for any belief state.

3.4.3 Improvement by adding memory

If all the previous methods fail to find an improvement then we try to improve the controller by adding more memory. This will break the controller out of the local optimum and then we can continue with the least expensive steps again.

3.4.3.1 Platzman escape technique

The method is described in the previous chapter. The technique checks whether there are any stochastic nodes in the controller which are represented as a convex combination of more than one deterministic node. If this is the case, then we can add one of those deterministic nodes to the controller, and normalize the event probabilities for stochastic nodes having non-zero probability for that deterministic component. A point worth mentioning here is that we are assuming that in the previous step we have already performed derandomization. This is important; otherwise there is a possibility of adding a duplicate node.

3.4.3.2 Global search using branch and bound algorithm

If all previous methods fail then we perform the branch and bound algorithm described in Section 3.3, to find the node that gives maximum improvement for any belief state. We can also try some of the variations of the algorithm described in the section on the branch and bound algorithm. If we don't find an improvement then that guarantees an ϵ-optimal controller.

1. Input : An initial stochastic finite state controller δ, a parameter ϵ providing bound on optimality of the controller.

2. Policy evaluation: Using Equation 2.5 compute the value function V^{δ} for controller δ.

3. Policy Improvement by removing dominated nodes:

   (a) Remove pointwise dominated nodes

   (b) Remove jointly dominated nodes

   (c) If none of the nodes were changed in step 3(a) or 3(b) then goto step 4. Else prune any memory state of δ which is not reachable from any other vector in δ, goto step 2.

4. Policy Improvement without adding memory

   (a) Improve policy by de-randomization. If none of the nodes were changed then goto step 4(b). Else, goto step 2.

   (b) Improvement by randomization:

      i. For each memory state in δ, use the linear program from Table 2.6 with set of variables obtained from Table 3.1 to improve the value function associated with it for each state. Change the action and node transition probabilities associated with that memory state to those that correspond to improved values.

      ii. If the linear program achieves some improvement for any memory state, then modify δ and go to step 2, else go to step 5.

5. Policy improvement by adding memory:

   (a) Efficient escape technique : If there are any stochastic nodes in the controller which are represented as a convex combination of more then one deterministic node. Then add one of those deterministic nodes to the controller, and normalize the event probabilities for stochastic nodes having non-zero probability for that deterministic component. Goto step 2.

   (b) Global search using branch and bound algorithm : If all the nodes in the current controller are deterministic, perform the branch and bound algorithm described in Section 3.3, to find the node that gives maximum improvement for any belief state (or alternatively find a node that gives improvement for any belief state).

   (c) Termination test: Calculate the Bellman residual from the improvement obtained using branch and bound, if it is less than or equal to $\epsilon(1-\beta)/\beta$, then go to step 6. Else set δ to δ' and go to step 2.

6. Output : An ϵ -optimal stochastic finite state controller.

Table 3.2 Stochastic Policy Iteration

CHAPTER IV

RESULTS

4.1 Overview

This chapter presents some results of testing the techniques described in the previous chapter. The test results are for five benchmark POMDPs used widely in the literature. Some characteristics of the problems are described in Table 4.1.

Problem	Num States	Num Actions	Num observations
Tiger	2	3	2
Cheese	11	4	7
Network	7	4	2
Aircraft	12	6	5
Shuttle	8	3	5

Table 4.1 Benchmark problems

S.No.	Problem	largest LP	vars in ILP	largest ILP	vars in LP
1	Tiger	1620	76	76	1620
2	Cheese	1316	35	42	1232
3	Network	2256	150	150	2256
4	Shuttle	3855	111	111	3855
5	Aircraft	2520	618	618	2520

Table 4.2 Reduction in the number of variables

4.2 Reduction in the number of variables

Table **??** shows how much the simplification technique reduces the number of variables in Poupart's LP. These problems have varying degrees of hardness, and therefore we tested these problems with different error bounds. The column labeled ”vars in LP” shows the number of variables in the largest linear program solved for that problem. The column labeled ”vars in ILP” shows the number of variables remaining after applying the reduction technique for that instance. The next two entries in the table show the number of variables in the largest linear program that the simplified linear program had to solve, and the corresponding number of variables in the original linear program. As these results show, the simplification technique dramatically reduces the size of the LP.

4.3 Pruning using branch and bound

Table 4.3 shows the amount of pruning achieved by the branch and bound algorithm.

Problem	Max num LP	Actual number of LP solved
Tiger	217083	1299
Cheese	313457000000	49
Network	287296	1708
Aircraft	23634200000	9234

Table 4.3 Pruning using branch and bound

The column labeled ”max num LP” shows the maximum number of deterministic nodes that could be added to the controller, and corresponds to the number of linear programs that would need to be solved in enumerating all possibilities in searching for the best node. The last column shows the number of linear programs actually solved by the branch-and-bound algorithm. The results are for the largest size controller for each instance. They clearly show the effectiveness of pruning.

4.4 Timing results

Table 4.4 shows the relative time taken by using Poupart’s LP to improve a controller without adding a node, compared to the time taken by branch and bound to add a node, in solving each of these problems.

S.No.	Problem	Time taken by linear program	Time taken by branch and bound
1	Tiger	2134	615
2	Cheese	446	11
3	Network	13863	1610
4	Shuttle	15495	1156
5	Aircraft	3072	1800

Table 4.4 Timing results for the algorithm (in CPU seconds)

CHAPTER V

CONCLUSION AND FUTURE WORK

This thesis considers a bounded policy iteration algorithm for POMDPs that improves a policy that takes the form of a stochastic finite-state controller, and presents two improvements of this algorithm. The first improvement provides a simplification of the basic linear program, which is used to find improved controllers. This results in a considerable speed-up in efficiency of the original algorithm. The second enhancement concerns the method used to break out of the local optimum by adding a node to the controller. A branch and bound algorithm is presented that adds the best possible node to the controller, and also provides an error bound and a test for global optimality.

Several additional improvements seem possible, and would be investigated in future. Some of them are mentioned as follows:

1. A new improved linear program: In the current linear program in Table 2.6, we try to find a pointwise improvement for each node. This imposes tough constraints on the linear program. We are analyzing a new linear program that will have more relaxed constraints.

2. It also seems possible to find heuristic functions that will provide a tighter upper bound than the present heuristic function used.

3. Finally, these techniques can be applied to a bounded policy iteration algorithm for multi-agent POMDPs [2].

REFERENCES

[1] J. Baxter, *Gradient-Based Learning of Controllers with Internal State*, technical report, Research School of Information Science and Engineering, Australian National University, Australia, 2000.

[2] D. Bernstein, E.Hansen, and S. Zilberstein, "Bounded Policy Iteration for Decentralized POMDPs," *To appear in Proceedings of the 19th International Joint Conference on Artificial Intelligence(IJCAI-05)*, 2005.

[3] A. Cassandra, M. Littman, and N. Zhang, "Incremental Pruning: A Simple, Fast, Exact Method for Partially Observable Markov Decision Processes," *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–97)*, D. Geiger and P. P. Shenoy, eds., San Francisco, CA, 1997, pp. 54–61, Morgan Kaufmann Publishers.

[4] E. Hansen, *Finite Memory Control of Partially Observable Systems*, doctoral dissertation, Department of Computer Sciences, University of Massachusetts, Amherst, Massachusetts, 1998.

[5] E. Hansen, "Solving POMDPs by Searching in Policy Space," *Proceedings of the eighth International Conference on Uncertainty in Artificial Intelligence*, Madison, WI, 1998, pp. 211–219.

[6] J. Kim, "A Hierarchical Approach to Probabilistic Pursuit Evasion Games with Unmanned Ground and Aerial Vehicles," *Proceedings of the 40th International Conference on Decision and Control*, 2001.

[7] M. Littman, T. Dean, and L. Kaelbling, "On the Complexity of Solving Markov Decision Problems," *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*. 1995, AAAI Press.

[8] M. L. Littman, "Memoryless policies: Theoretical limitations and practical results," *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, D. Cliff, P. Husbands, J.-A. Meyer, and S. W. Wilson, eds., Cambridge, MA, 1994, The MIT Press.

[9] N. Meuleau, K. Kim, L. Kaelbling, and A. Cassandra, "Solving POMDPs by Searching the Space of Finite Policies," *Proceedings of the Fifteenth Conference on Un-*

certainity in Artificial Intelligence, San Francisco, CA, 1999, pp. 284–292, Morgan Kaufmann.

[10] L. Platzman, *Finite-memory Estimation and Control of Finite Probabilistic Systems,* doctoral dissertation, Massachusetts Institute of Technology, Massachusetts, 1977.

[11] L. K. Platzman, *A Feasible Computational Approach to Infinite-Horizon Partially-Observed Markov Decision Problems,* technical report, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia, 1981.

[12] P. Poupart and C. Boutlier, "Bounded Finite State Controllers," *In Proceedings of the Conference on Neural Information Processing Systems(NIPS-03)*, 2003.

[13] N. Roy, J. Pineau, and S. Thrun, "Spoken Dialogue Managment using Probabilistic Reasoning," *Proceedings of the 38th Annual Meeting of the Association of Computational Linguistics*, 2000.

[14] N. Roy and S. Thrun, "Coastal navigation with mobile robots," 1999.

[15] S. P. Singh, T. Jaakkola, and M. Jordan, "Learning Without State-Estimation in Partially Observable Markovian Decision Processes," *Proceedings of the Eleventh Machine Learning Workshop,* 1994, pp. 284–292.

[16] E. Sondik, *The Optimal Control of Partially Observable Markov Decision Processes,* doctoral dissertation, Stanford University, Stanford, California, 1971.