# THE ROLE OF REPRESENTATION AND ABSTRACTION IN STOCHASTIC PLANNING

A Dissertation Presented

by

ZHENGZHU FENG

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

DOCTOR OF PHILOSOPHY

September 2005

Computer Science

# THE ROLE OF REPRESENTATION AND ABSTRACTION IN STOCHASTIC PLANNING

A Dissertation Presented

by

ZHENGZHU FENG

Approved as to style and content by:

[illegible]

Shlomo Zilberstein, Chair

[illegible]

Andrew G. Barto, Member

[illegible]

Sridhar Mahadevan, Member

[illegible]

Norman Sondheimer, Member

[illegible]

W. Bruce Croft, Department Chair
Computer Science

## ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Shlomo Zilberstein, for his support during my years as a PhD student. His encouragement and guidance were indispensable in the completion of this dissertation.

I would like to thank my collaborators. Eric Hansen, who was my advisor when I was working on my Master's degree at Mississippi State, introduced me to the world of MDPs and POMDPs. He provided early guidance in pursuing the issues of representation and abstraction in these models. Richard Dearden, Nicolas Meuleau and Rich Washington provided great help when I was working as an intern at NASA/Ames. Together we developed the ideas on state abstraction in continuous MDPs. These collaborations helped shape my ideas on how to perform abstraction for the more general POMDP models.

I thank members of my committee. Discussions with Sridhar Mahadevan helped clarify my thoughts and improved the structure of the dissertation. Andy Barto and Norman Sondheimer provided detail comments on the draft which helped improve its quality significantly.

Finally, I thank my family for their great support and patience during my years as a graduate student.

## ABSTRACT

## THE ROLE OF REPRESENTATION AND ABSTRACTION IN STOCHASTIC PLANNING

SEPTEMBER 2005

ZHENGZHU FENG

B.S., SOUTH CHINA UNIVERSITY OF TECHNOLOGY

M.S., MISSISSIPPI STATE UNIVERSITY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Shlomo Zilberstein

Markov decision process (MDP), originally studied in the Operations Research (OR) community, provides a natural framework to model a wide variety of sequential decision making problems. Because of its powerful expressiveness, the AI community has adopted the MDP framework to model complex stochastic planning problems. However, this expressiveness in modeling comes with a hefty price when it comes to solving the MDP model and obtaining an optimal plan. Scaling up solution algorithms for MDPs is thus a critical research topic in AI that has received a lot of attentions.

In this thesis I study the role of representation and abstraction in scaling up solution methods for various MDP models. Three variants of MDP models are studied in this thesis: A discrete state, fully observable model; a continuous state, fully observable model; and a discrete state, partially observable model. One contribution of this thesis is the development of new algorithms for these models that use various representations to exploit natural state abstractions. These new algorithms significantly increase the range of problems that can be solved in practice.

A second contribution is the formulation of a new type of belief-space structure in partially observable MDPs. Using a region-based representation, new algorithms are able to reduce the computational time exponentially while still maintaining the optimality of the solution. This presents a breakthrough in scalability studies for this model. The results open up a range of opportunities to gain better understanding of the model and its complexity, and to develop better computational solutions.

# TABLE OF CONTENTS

## LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

Automated planning has many applications in a wide variety of areas such as manufacturing, robotics and space exploration. Research in classical planning is based on simplifying assumptions about the capabilities of the agent as well as the planning task. Typically, agents have perfect actuators and sensors, and they interact with the environment in a fully predictable, deterministic way. The objective of the agents is usually to reach certain *goal* states. Although there are many applications that can satisfy these assumptions, many more cannot. In real-world situations, agents must cope with various uncertainties and complex objectives.

One source of uncertainty is in the outcome of actions that the agent can take along the plan. Some actions may not have the intended effect because of various physical constraints on the agent's capabilities. For example, a robot picking up a part may fail to hold it from time to time. Other actions may by inherently stochastic. For example, performing a scientific test may produce various results. Another source of uncertainty is in the limited ability of the agent to accurately identify the current state of the system. Because of the uncertainty of the outcome of actions, there are multiple possible resulting states that the agent can be in after an action is taken. It is crucial that the agent can identify the exact outcome of such an action. In practice this is not always possible because of the limited sensing capability that the agent has. Finally, for real life applications there may not be a well defined goal state. Instead, the performance of any plan may be evaluated by some general utility function, and the objective is to maximize the utility. This thesis addresses the problem of optimal planning under these circumstances.

## 1.1 Decision theoretic planning

A Markov decision process (MDP) provides an attractive framework for modeling such complex planning problems. In its most general form, an MDP can model uncertainties in both the agent's actuators and the sensors, and can model both goal oriented and general utility oriented planning tasks.

Research in MDPs can be traced back to the 1950's when Bellman studied the dynamic programming paradigm in sequential decision making [6]. Since then a large literature on the topic has been formed, mainly by the operations research and control theory communities [62, 39, 2, 95, 79, 74]. Because of its powerful expressiveness, the AI community has adopted the MDP framework to model complex planning and learning problems. However, this expressiveness in modeling comes with a hefty price when it comes to solving an MDP and obtaining an optimal plan. Scaling up planning algorithms for MDPs is thus a critical research topic that has received a lot of attentions [35, 22, 66, 98, 26, 52, 58].

Exact solution to an MDP in general requires computing a value function over the state space of the system. For fully observable MDPs with discrete state space, the number of states grows exponentially with the number of features describing the system. Therefore although algorithms for this type of MDPs are generally considered tractable [83], they do not scale to large state spaces. For MDPs with continuous state spaces, as well as the more general *partially observable* MDP (POMDP) model, the state space over which a value function must be computed is uncountably infinite. This large or infinite state space is therefore the main source of computational complexity in solving MDPs. In fact, solving finite horizon POMDPs is intractable assuming P$\neq$ PSPACE [83, 75], and solving infinite horizon POMDPs is undecidable [76].

Note however that these complexity results are for the worst case scenarios. In practice, many problems exhibit various structures that make the problems easier to solve. Therefore a critical key to scalable algorithms for MDPs is to discover, understand, and exploit these structures. This thesis will demonstrate various kinds of abstract structures in different models and domains, explain how they interact with the solution process, and present scalable algorithms to take advantage of these structures.

## 1.2 Representation and abstraction

Abstraction is the hallmark of human problem solving. Being able to ignore irrelevant details at the proper level helps people to cope with the complexity of problem solving. Imagine if we had to reason about all the details of our everyday life, we would never be able to accomplish anything. There are many useful forms of abstractions. The focus of this thesis is on the notion of *state abstraction* in connection to the large and sometimes infinite state spaces of various MDP models. In this sense, an abstraction means grouping states to form *abstract states* that can be treated as single entities for our computational needs. Equivalently, an abstraction can also be seen as a partitioning of the state space, where each partition is an abstract state.

The process of forming abstractions involves analyzing the state space and identifying groups of states that can be treated the same for the current reasoning task. The representation of the state space plays a crucial role in how efficient and useful such abstraction can be. In the simplest case, the state space can be represented by a list of all unique states. Forming abstraction can be accomplished by walking through this list and marking states that can be treated the same. However, the cost of forming abstraction using this so called *flat* representation is usually too great, because it involves enumerating the large state space, which is the cause of inefficiency we want to avoid at the beginning. Further, for uncountable state spaces, enumeration becomes impossible. The central theme of this thesis is on how to exploit various representations to form abstractions effectively and efficiently for different types of state spaces, so that the underlying computation can benefit the most from the abstractions.

## 1.3 Exact and approximate algorithms

A general form of state abstraction is function approximation, which uses simple functions such as polynomials or neural networks to approximate the true value of all states in the state space, thus avoiding explicitly enumerating the state space. It is relatively easy to control the complexity of approximation algorithms by choosing functions that are known to be easy to compute. As a result, many researchers have focused on approximation algorithms for MDPs [39, 74, 84, 10, 22, 19, 56, 57, 103, 86, 89]. These studies are very important in that they identify specific instances of problems where approximation algorithms deliver satisfactory results. On the other hand, most approximation algorithms cannot provide tight bounds on the approximation error. Some approximation algorithms can in fact produce arbitrarily poor results, as pointed out in refs. [51, 21]. Further, most approximation algorithms rely on some strong assumptions about the domain, and it is not clear how well they can be generalized to other domains.

This thesis focuses on exact solution algorithms for various MDP models. Besides retaining sound theoretical properties, a better exact algorithm can help us better understand the general structure of the problem, through the process of solving the problem completely, and from the optimal solution revealed in this process. With a better understanding of the general structure, we can in turn develop approximate algorithms with better theoretical properties, as we did in refs. [43, 45].

## 1.4 Thesis

The main thesis of this work is that there are useful and general abstract structures in MDP and POMDP models that can be represented and exploited to significantly accelerate exact solution algorithms. This is supported by rigorous theoretical analysis and substantial empirical evaluations. Many results in this thesis have been applied and verified in different domains and settings by other researchers.

## 1.5 Contributions

The main contributions of this thesis are the discovery and analysis of various representation and abstraction schemes for several important MDP models, and the design and analysis of algorithms that exploit these representations and abstractions. It not only contributes state-of-the-art algorithms for these models, but also improves the general understanding of their special structures and their relations to scalability.

Previous research on state abstraction for MDPs has focused on a simplified but nevertheless important MDP model where the state space is assumed to be finite and the agent has full observability over the finite state space. Two representations have been developed to facilitate state abstraction for this case: the decision tree representation [15, 37, 16] and the decision diagram representation [59, 97]. In particular, the decision diagram approach borrows data structures developed in the field of automated model checking [32] that have been shown to be particularly effective in forming state abstraction for the above simplified MDP model.

Chapter 3 extends this research by combining the symbolic representation with search algorithms for discrete state MDPs in a fully integrated symbolic model checking framework. The resulting algorithm outperforms existing algorithms by orders of magnitudes on public benchmark problems, and won a first place award in the probabilistic track of the 2004 International Planning Competition [73].

Chapter 4 introduces a new geometric representation for fully observable MDPs with continuous state spaces. Abstraction over the continuous state space is formed by partitioning the continuous space into rectangular regions, where the value over each region can be a constant or a piece-wise linear function. The resulting algorithm effectively discretizes the continuous space on-demand, spending computational resource where it is needed. It has been applied to a protocol Mars rover model developed at NASA and outperformed previous approaches by orders of magnitudes. This research is being continued at NASA with an attempt to deploy the algorithms on physical rover platforms [78, 77].

For the more general POMDP model, state abstraction is considerably more difficult because the solution of a POMDP involves a transformation to a special continuous state space called a *belief space* that does not lend itself to the rectangle partition representation developed in Chapter 4, or any other regular partitioning scheme. In fact, the representation of the value function over this belief space is the major difficulty in POMDP algorithms. Chapter 5 develops an abstraction algorithm for the original discrete state space of a POMDP, which translates into reduced dimensionality of the transformed belief space. It represents the first algorithm to explicitly exploit state abstraction for POMDPs. Because of the generality of the abstraction scheme, it has been applied to other POMDP algorithms [54].

In Chapter 6, a new form of abstraction for the belief state space of POMDPs is introduced. Using a region-based representation, the notion of abstraction of the belief space is being made explicit. By exploiting a region-based abstraction, the computational time for the cross-sum operation, a central bottleneck in POMDP algorithms, can be reduced exponentially. This exponential reduction in turn exposes another bottleneck in POMDP algorithms: the maximization step. The same region-based representation can also be used to exploit abstraction in speeding up the maximization step. The algorithms developed in this chapter represent the current state-of-the-art exact algorithms for solving POMDPs.

# CHAPTER 2

# MARKOV DECISION PROCESSES

This chapter describes the Markov decision process model and reviews standard algorithms for solving it. I will focus on the scalability issues of these standard solution algorithms and their relation to state space representations and abstractions. No new algorithm is introduced in this chapter.

## 2.1 Basic model

A Markov decision process (MDP) is defined as a tuple $(S, A, P, R, \beta)$ where:

- $S$ is a set of states;
- $A$ is a finite set of actions;
- $P$ is the transition model. $P^a(s'|s)$ specifies the probability of reaching state $s'$ when action $a$ is taken in state $s$. $s$ is usually referred to as the *pre-action* state, and $s'$ the *post-action* state;
- $R$ is the reward model. $R^a(s)$ specifies the expected reward for taking action $a$ in state $s$.
- $\beta$ is a discount factor, where $0 \leq \beta \leq 1$.

In this definition, the Markovian property is assumed implicitly by the fact that the transition probability and the reward of an action only depends on the current state. An MDP models an agent acting in a dynamic system whose state space is $S$. In this thesis, two types of state spaces are studied: finite discrete state space, and uncountably infinite state space.

For the moment, the two cases are not distinguished. The basic theory holds for both cases as long as the transition and reward model are well defined over the state space.

The basic model assumes full observability of the agent, that is, the agent can accurately identify the state it is in at any time step. At each time step $t$, the agent observes the current state $s_t$, makes a decision to take action $a_t$, the system then transfers to the next state $s_{t+1}$ according to the transition model $P^{a_t}$. At the same time the agent receives a reward $R^{a_t}(s_t)$ according to the reward model. In general, the decision making at time step $t$ is controlled by a policy $\pi(s,t)$ that maps the $t$-step states to actions. The objective of the agent is to maximize the expected total reward acting in the system:

$$\max_{\pi} E \sum_{t=0}^{T} \beta^t R^{\pi(s_t,t)}(s_t).$$

Here $E$ represents the expectation over all possible state trajectories. $T$ is the *planning horizon*, that is, the number of steps that the agent is allowed to take before the process is terminated. When $T$ is finite, the problem is referred to as a finite horizon problem. When $T$ is infinite, it is referred to as an infinite horizon problem. For infinite horizon problems, the discount factor $\beta$ has to be strictly less than 1 to ensure that the sum of rewards is finite.

## 2.2 Dynamic programming

Dynamic programming is the standard algorithm for solving MDPs. Define value function $V^n(s)$ to be the value obtainable if the agent is in state $s$ and there are $n$ steps left to be taken. When $n = 0$, that is, no more action is allowed, the value is 0 for all states: $V^0(\cdot) = 0$. Given the $n$-step value function $V^n$, the $(n+1)$-step value function is computed using the following dynamic programming (DP) update:

$$V^{n+1}(s) = \max_{a \in A} \left\{ R^a(s) + \beta \sum_{s' \in S} P^a(s'|s) V^n(s') \right\} \tag{2.1}$$

This update needs to be computed for all states $s \in S$. In the following, we assume that when computing $V^k(s)$, $a^k(s)$ is the action that maximizes the right-hand-side of the above equation.

For finite horizon problem with a planning horizon of $T$, we need to compute the DP update from $V^1$ up to $V^T$. The optimal $T$-step policy is simply:

$$\pi^*_T(s,t) = a^{T-t}(s).$$

Note the step index of the DP update is in reverse of the time step index of the agent's actions.

For infinite horizon problems, it can be shown that the DP update will converge in the limit to the optimal value function:

$$\exists V^* s.t. V^* = \lim_{n \to \infty} V^n$$

In this case, the optimal policy is also *stationary*, and can be extracted from $V^*$ by an additional step of DP update:

$$\pi^*(s) = \arg\max_{a \in A} \left\{ R^a(s) + \beta \sum_{s' \in S} P^a(s'|s) V^*(s') \right\}$$

In practice, only a finite number of DP updates can be computed. The distance between the $n$-step value function and the optimal infinite horizon value function can be bounded:

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

The distance $\max_s |V^{n+1}(s) - V^n(s)|$ is usually called the Bellman residual.

The above process for solving infinite horizon MDPs is usually called *value iteration* (VI), since it iteratively improves a value function until it is good enough, and then extract the control policy from it. Another algorithm for solving infinite horizon MDPs is the *policy iteration* (PI) algorithm, which iteratively improves a policy directly. Given stationary policy $\pi$, its value $V^{\pi}$ can be computed by solving the following linear system in the *policy evaluation* step:

$$V^{\pi}(s) = R^{\pi(s)}(s) + \beta \sum_{s' \in S} P^{\pi(s)}(s'|s)V^{\pi}(s'), \forall s \in S \tag{2.2}$$

After the evaluation, PI performs a *policy improvement* step, which extracts a greedy policy from the value of the previous policy:

$$\pi'(s) = \arg\max_{a \in A} \left\{ (R^{a}(s) + \beta \sum_{s' \in S} P^{a}(s'|s)V^{\pi}(s'). \right\} \tag{2.3}$$

Note the policy improvement step is essentially a DP update. This process then repeats with the current policy replaced by $\pi'$. It can be shown that the sequence of policies produced by this process converges to the optimal policy.

As we can see, both value iteration and policy iteration make use of the DP update in Equation 2.1, which is also the main computation in both algorithms. The algorithms described in the rest of the thesis are all focused on computing the DP update more efficiently. They are applicable to both value iteration and policy iteration.

## 2.3 Partially observable models

In an fully observable MDP, the agent can accurately identify the current state. In a partially observable MDP (POMDP), this assumption is removed: the agent can only "guess" what the current state is based potentially noisy sensor inputs. Formally, a POMDP is defined by the tuple $(S, A, P, R, Z, O, \beta)$, where $A, P, R$ and $\beta$ are the same as in the MDP model described in the previous section, and

- $S$ is a finite set of states;

- $Z$ is a finite set of observation states;
- $O$ is the observation model. $O^a(z|s')$ is the probability that $z$ is observed if action $a$ is taken and resulted in state $s'$;

Note that unlike the basic model, the state space of a POMDP is assumed to be finite.

Allowing partial observability makes POMDPs substantially more expressive then MDPs. However, to obtain the optimal solution of a POMDP, one has to rely on an equivalent MDP model. The standard solution method is to convert a POMDP to a fully observable MDP over *belief states*. A belief state $b$ is a probability distribution over the discrete state space $S$:

$$b : S \to [0, 1],$$

such that

$$\sum_{s \in S} b(s) = 1 \; .$$

Figure 2.1 shows the belief state space of a 2-state POMDP on the left, and a 3-state POMDP on the right. In the 2-state case, the belief space is a line between the two states. A belief state $b$ is a point on this line, whose distance to $s_0$ is the probability of being in $s_1$ and vice verse. In the 3-state case, the belief space is a triangle on a plane. A belief state $b$ is a point on this plane, whose distance to the edge of the triangle opposing $s_i$ is the probability of being in state $s_i$.

Given a belief state $b$ representing the agent's current best estimate of the underlying state, there are at most $|Z|$ possible resulting belief states after action $a$ is taken, one for each possible observation. The probability of seeing a particular observation $z$ can be computed from the POMDP model:

$$P^a(z|b) = \sum_{s' \in S} \left[ O^a(z|s') \sum_{s \in S} P^a(s'|s) b(s) \right] . \tag{2.4}$$

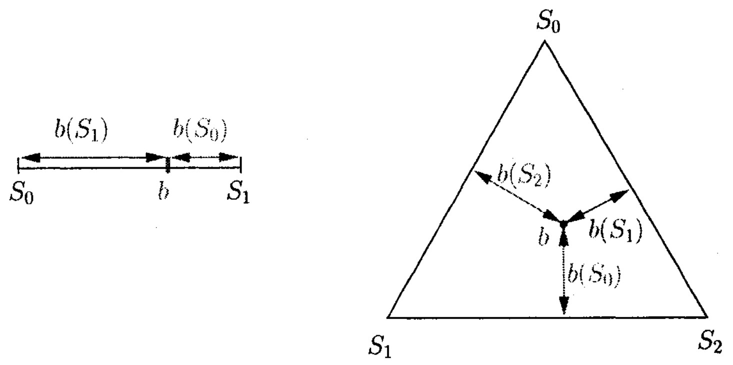


**Figure 2.1.** Belief state space for 2-state and 3-state POMDPs.

The resulting belief state $b_z^a$ can be computed using Bayesian conditioning as follow:

$$b_z^a(s') = \frac{1}{P^a(z|b)} O^a(z|s') \sum_{s \in S} P^a(s'|s) b(s). \tag{2.5}$$

We use $b_z^a = T_z^a(b)$ to refer to belief update. It can be shown that a belief state updated this way is a sufficient statistic that summarizes all the previous history of the process [92]. It is the only information needed in choosing an optimal action. An equivalent, fully observable MDP can be defined over this belief state space as the tuple $(B, A, T, R_B)$, where

- $B$ is the space of belief states;
- $A$ is the action set as before;
- $T$ is the transition function as defined above. If action $a$ is taken in belief state $b$, then the system transfers to belief state $T_z^a(b)$ with probability $P^a(z|b)$;
- $R_B^a$ is the reward for taking action in belief state $b$, computed from the POMDP model: $R_B(b) = \sum_{s \in S} b(s) R^a(s)$.

As in the fully observable case, we can perform dynamic programming for a POMDP using the equivalent belief-state MDP. Define value function $V : B \longrightarrow \Re$. Then the DP update for POMDP is as follow:

$$V^{n+1}(b) = \max_{a \in A} \left\{ R_B^a(b) + \beta \sum_{z \in Z} P^a(z|b) V^n(T(b)) \right\}, \forall b \in B \tag{2.6}$$

With this, algorithms such as value iteration and policy iteration can be constructed for POMDPs in exactly the same way as in the basic MDPs, provided that the value function over the now uncountably infinite belief space can be represented exactly. The representation of the value function will be reviewed in Section 2.6.

## 2.4 Computational complexity

The computational complexity of solving MDP models is closely related to their state spaces. Discrete state fully observable MDPs are generally considered as *tractable* problems, since they can be solved in polynomial time and space in the size of the state space [83]. However, the size of the state space is usually exponentially large for practical problems. Therefore algorithms that rely on enumerating the state space are generally considered *unscalable*. A survey on complexity results for MDPs can be found in [72].

For POMDPs, or equivalently, belief-state MDPs, the complexity is much higher. Solving finite horizon POMDPs is *intractable* assuming P$\neq$ PSPACE [83, 75], and solving infinite horizon POMDPs is *undecidable* [76]. Algorithmically, the uncountably infinite belief space is the main source of computational complexity in solving POMDPs.

Note that the complexity results reflects the worst case scenario. In practice, many problems exhibit special structures that allow more efficient algorithms to be developed, provided that these structures can be represented and manipulated efficiently. In the following sections I review previous research in this regard.

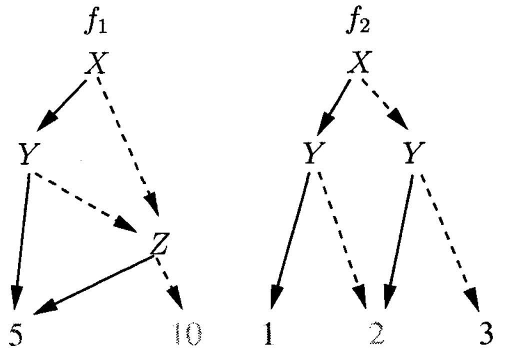
## 2.5 Symbolic representation for discrete state MDPs

Traditionally, the state space of a discrete state MDP is represented by enumerating all the distinct states in the system. It is usually stored in the computer as a table or a list. The transition model is represented by a matrix with one entry for each pair of pre-action and post-action states. The reward function and the value function used in dynamic programming is stored as a table with one entry for each state. This representation is sometimes called a *flat* representation [14]. In practice, however, richer representations are often used to model a problem domain. One common rich representation is that of a propositional, or symbolic representation, in which the state space is described using a set of Boolean state variables. A unique state in such a system is a full instantiation of the state variables. To model such domains as MDPs, a conversion from the richer representation is usually carried out by enumerating all the combinations of the state variables. The transition and reward functions are then constructed on top of this enumerated state space. This leads to the state explosion problem, where the number of states increases exponentially as the number of state variables increases. This not only increases the cost of *representing* the problem, but more importantly the cost of *solving* the problem.

On the other hand, direct manipulation of a symbolic representation is possible and proved useful in many other areas. For example, research in single step decision making such as Bayesian network [85] and influence diagram [63] use state variables to describe the state space of a problem. They also use graphs to compactly describe the relations among the state variables, exploiting the sparse nature of the dependence relations in these systems. Outside the area of Artificial Intelligence, symbolic representation has been extensively studied in the field of model checking [24, 32, 65]. Efficient data structures and algorithms have been developed to manipulate functions over Boolean variables. Of particular interest to this thesis are the *Binary decision diagrams* (BDDs) [24] and *Algebraic decision diagrams* (ADDs) [4].

### 2.5.1 Algebraic and binary decision diagrams

ADDs are a generalization of BDDs, a compact data structure for Boolean functions used in symbolic model checking. A decision diagram is a data structure that corresponds to an acyclic directed graph. It compactly represents a mapping from a set of Boolean state



|  | $XYZ$ | $f_1$ | $f_2$ |
|---|---|---|---|
| $S_0$ | 111 | 5 | 1 |
| $S_1$ | 110 | 5 | 1 |
| $S_2$ | 101 | 5 | 2 |
| $S_3$ | 100 | 10 | 2 |
| $S_4$ | 011 | 5 | 2 |
| $S_5$ | 010 | 10 | 2 |
| $S_6$ | 001 | 5 | 3 |
| $S_7$ | 000 | 10 | 3 |

**Figure 2.2.** ADDs and state abstraction.

variables to a set of values. A BDD represents a mapping to the values 0 or 1. An ADD represents a mapping to any finite set of values. To represent these mappings compactly, decision diagrams exploit the fact that many instantiations of the state variables may map to the same value. In other words, decision diagrams exploit state abstraction.

Figure 2.2 shows two ADDs representing two functions $f_1$ and $f_2$ over three state variables $X, Y$ and $Z$. Each node in an ADD represents a state variable. A solid edge from a node represents the *true* value of that node, and a dashed edge represents the *false* value. To find the function mapping of a particular instantiation of the variables, one traverses from the root of the diagram following the edges according to the truth assignment in that particular instantiation, until a leaf node is reached. The value in the leaf node is the value that the function assigns to that particular variable instantiation.

The corresponding flat representation of the same functions are listed in the table on the right of Figure 2.2. As we can see, although there are 8 unique states in the state space, some of the states maps to the same value. The ADD representation captures this structure, by ignoring the state variables when they are irrelevant. As a result, the ADD representation for the two functions are much more compact than the table representation.

In addition to representing functions more compactly, ADDs and BDDs can be used to perform computation such as summation and multiplication over functions more efficiently. Figure 2.3 shows an example of adding two functions represented by two ADDs. Instead

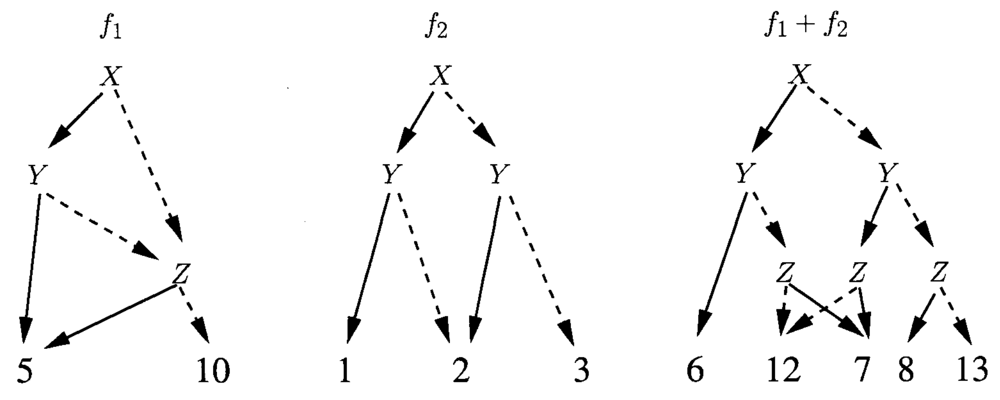


**Figure 2.3.** Computation using ADDs.

of converting the diagrams representation of the functions into tables, one can recursively traverse the diagrams to construct a new diagram that represents the resulting function. Efficient algorithms and caching schemes for manipulating decision diagrams and performing computations have been developed by the symbolic model checking community. There are professional software packages publicly available [69, 93].

## 2.5.2 Symbolic dynamic programming for MDPs

Hoey *et. al*[59] describe how to represent the transition and reward models of a MDP compactly using ADDs. Let $\mathbf{X} = \{X_1, \ldots, X_n\}$ represent the state variables at the current step, and let $\mathbf{X}' = \{X'_1, \ldots, X'_n\}$ represent the state variables at the next step. For each action, an ADD $P^a(\mathbf{X}, \mathbf{X}')$ represents the transition probabilities for the action.[1] Similarly, the reward model $R^a(\mathbf{X})$ for each action $a$ is represented by an ADD, so is the value function $V(\mathbf{X})$. The advantage of using ADDs to represent mappings from states (and state transitions) to values is that the complexity of operations on ADDs depends on the size of the diagrams, not the size of the state space. If there are sufficient regularities in

[1]In practice, constructing the full transition function $P^a(\mathbf{X}, \mathbf{X}')$ may not be feasible. Hoey *et. al*[59] discuss how to construct the full transition function incrementally while performing the dynamic programming update. The technique is used in implementing algorithms presented in Chapter 3.

the model, ADDs can be very compact, allowing problems with large state spaces to be represented and solved efficiently.

Using this representation, the DP update in Equation 2.1 can be carried out using ADD computations as

$$V^n(\mathbf{X}) = \max_{a \in A} \left\{ R^a(\mathbf{X}) + \beta \sum_{\mathbf{X}'} P^a(\mathbf{X}'|\mathbf{X}) V^{n-1}(\mathbf{X}') \right\}.$$

Here all the functions $V^n, V^{n-1}, P^a, R^a$ are represented by ADDs. The addition and multiplication operators are standard ADD arithmetic operators. The summation over $\mathbf{X}'$ is the so called *existential abstraction* operator in model checking, which effectively sums over values of all post-action states. This is basically the SPUDD algorithm [59]. Instead of enumerating the state space and computing Equation 2.1 once for each state, SPUDD computes an updated value function for all the states in one step using the various ADD operations. In doing so, it exploits the compactness of the ADD representation and the efficiency of ADD computations. Results reported in [59] suggest that SPUDD runs 30 to 40 times faster than traditional value iteration algorithms on nontrivial domains. For artificially constructed best case examples, SPUDD runs exponentially faster.

## 2.6 Implicit representation for belief-state MDPs

Unlike finite state MDP, the DP update for POMDPs (Equation 2.6) cannot be computed directly since the belief state space is uncountably infinite. Since $V(b)$ is a function defined over the infinite belief space, it is impossible to represent an explicit mapping from very belief state to values. Nevertheless, we will see that exact DP algorithms can still be carried out by exploiting the mathematical structures of the POMDP model. However, the lack of precise representation for the belief state space presents a major difficulty in exploiting state abstractions in POMDP research.

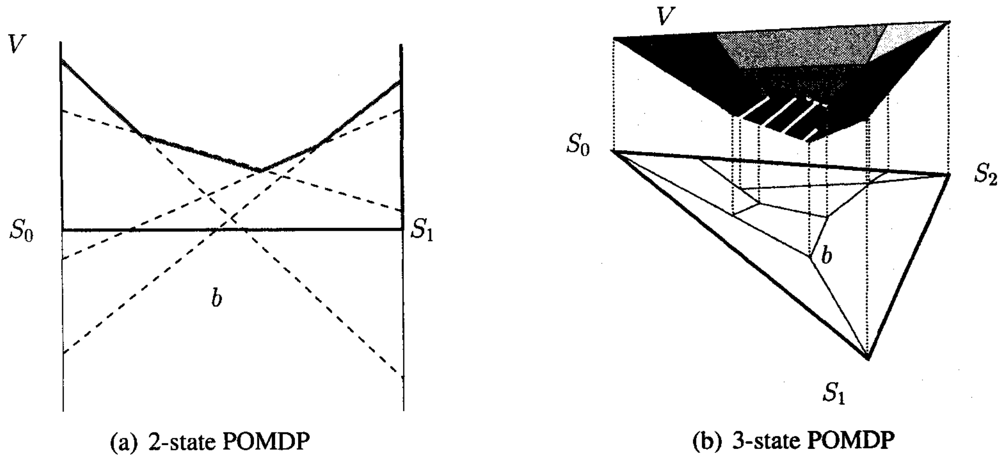


**Figure 2.4.** Sample value functions for POMDPs. (Source of picture on the right: [26])

### 2.6.1 Piece-wise linear and convex value function

The POMDP model posses special mathematical structures that allows the value functions used in the DP update to be represented implicitly without explicitly representing the belief state space. This is based on the following theoretical results:

1. The optimal value function for POMDP is convex [3];
2. For finite-horizon POMDPs, the value function is also piecewise linear [95, 92];
3. For infinite-horizon POMDPs, the optimal value function can be approximated arbitrarily closely by a piecewise linear and convex function [95, 96];
4. The DP update preserves the piecewise linear and convex (PWLC) property of the value function [95, 92].

Thus, if $V^n$ is piecewise linear and convex, $V^{n+1}$ as computed by the DP update is also piecewise linear and convex. We can represent a piecewise linear and convex function exactly using a finite set of vectors $\Gamma = \{\gamma_1, \ldots, \gamma_k\}$. Each vector $\gamma$ has a size of $|S|$. We use $\gamma(s)$ to denote the value of state $s$ in a particular vector $\gamma$. The value of a specific belief state $b$ can then be computed as

$$V(b) = \max_{\gamma_i \in \Gamma} b \cdot \gamma_i,$$

where

$$b \cdot \gamma := \sum_{s \in S} b(s)\gamma(s)$$

is the *dot product* between a belief state and a vector. Figure 2.4 shows examples of value functions of a 2-state POMDP and a 3-state POMDP.

With this representation, the DP update can be computed exactly by constructing a set of vectors that represents the updated value function $V^{n+1}$, given another set of vectors that represents the previous value function $V^n$. This process is usually broken down into several smaller steps [25]:

$$\begin{aligned} V^{a,z}(b) &= \frac{R_B(b)}{|Z|} + \beta P^a(z|b)V^n(T(b)); \\ V^a(b) &= \sum_{z \in Z} V^{a,z}(b); \\ V^{n+1}(b) &= \max_{a \in A} V^a(b). \end{aligned}$$

Each of these value functions is piecewise linear and convex, and can be represented by a set of vectors. Let $\mathcal{V}'$, $\mathcal{V}^a$, $\mathcal{V}^{a,z}$ and $\mathcal{V}$ be the vector sets representing $V^{n+1}$, $V^a$, $V^{a,z}$ and $V^n$, respectively. The DP update in terms of these vector sets is as follow:

$$\mathcal{V}^{a,z} = \{v^{a,z,i} | v^i \in \mathcal{V}\}; \quad (2.7)$$

$$\mathcal{V}^a = \oplus_{z \in Z} \mathcal{V}^{a,z}; \quad (2.8)$$

$$\mathcal{V}' = \cup_{a \in A} \mathcal{V}^a. \quad (2.9)$$

where $v^{a,z,i}$ is the vector defined by

$$v^{a,z,i}(s) = \frac{R^a(s)}{|Z|} + \beta \sum_{s' \in S} O^a(z|s')P^a(s'|s)v^i(s'), \tag{2.10}$$

Equation 2.10 is usually referred to as a *projection*: it projects vectors representing $V^n$ backward to obtain vectors that represents the expected value of executing action $a$ and observing $z$.

The $\oplus$ operator in Equation 2.8 is the so called *cross-sum* operator. Let $\mathcal{U}$ and $\mathcal{W}$ be two sets of vectors. The *cross sum* of $\mathcal{U}$ and $\mathcal{W}$ is defined as

$$\mathcal{U} \oplus \mathcal{W} = \{u + w | u \in \mathcal{U}, w \in \mathcal{W}\}.$$

Note the size of the cross-sum is the product of the size of the individual sets:

$$|\mathcal{U} \oplus \mathcal{W}| = |\mathcal{U}| \times |\mathcal{W}|.$$

This turns out to be the major source of complexity for POMDP algorithms. The vector representation of a $(n+1)$-step value function is exponential in the size of the $n$-step value function. Precisely:

$$|\mathcal{V}|^{n+1} \propto |A| \times |\mathcal{V}^n|^{|Z|}$$

Fortunately a large number of vectors can usually be pruned from this representation without affecting the value that it represents.

### 2.6.2 Pruning

For any PWLC function, there are infinitely many sets of vectors that can represent it. However, a PWLC function also has a unique and minimal set of vectors that represents it. Figure 2.5 shows two sets of vectors that represents the same PWLC function. The one on the right is the minimal representation. The DP update process in the previous section does

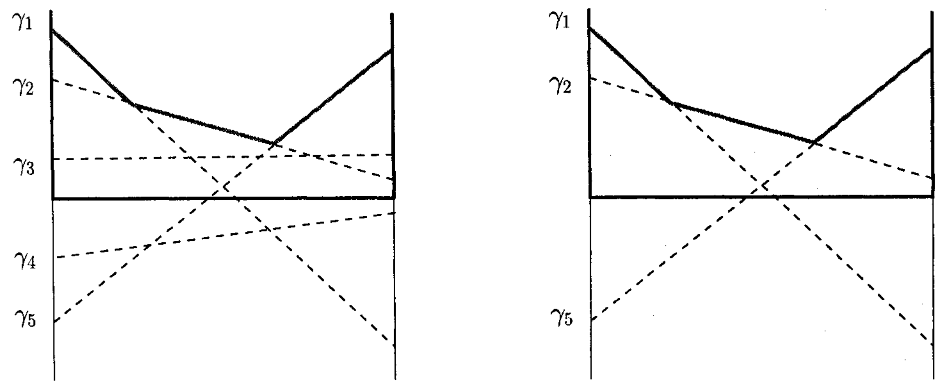


**Figure 2.5.** Two PWLC representations of the same value function.

not always generate minimal representations, and it is crucial to find the minimal representation before proceeding to the next step of DP update due to the exponential growth in the size of the vector sets at each step.

The minimal representation can be derived from a non-minimal representation by removing *dominated* vectors in that representation. The simplest method of removing dominated vectors is to remove any vector that is *point-wise dominated* by another vector. A vector $u$ is point-wise dominated by another vector $w$ if

$$u(s) \leq w(s) \text{ for all } s \in S$$

. The vector $\gamma_4$ in Figure 2.5 is an example of a point-wise dominated vector. It is dominated by the vector $\gamma_2$. Removing point-wise dominated vectors is relatively efficient, requiring a pair-wise comparison of vectors in a set. However, not all dominated vectors are point-wise dominated. For example, vector $\gamma_3$ in Figure 2.5 is not point-wise dominated by any other single vector; it is jointly dominated by $\gamma_2$ and $\gamma_5$.

There is a linear programming method that can detect all dominated vectors. Given a vector $u$ and a set of vectors $\mathcal{U}$ that does not include $u$, the linear program in Table 2.1 determines if $u$ is dominated or not. It searches through the whole belief space to find a

```
LPTEST($u, \mathcal{U}$)
variables: $d$, $b(s)\ \forall s \in S$
maximize $d$
subject to the constraints
    $b \cdot (w - u) \geq d,\ \forall u \in \mathcal{U}$
    $\sum_{s \in S} b(s) = 1$
```

**Table 2.1.** Standard linear program to test if a vector $u$ is dominated by a set of vectors $\mathcal{U}$.

belief point on which $u$ has the greatest distance $d$ from the upper surface defined by $\mathcal{U}$. If $d$ is positive, then $u$ is not dominated by $\mathcal{U}$. Otherwise it is dominated. This linear program has the following characteristics:

- The number of variables is $|S| + 1$,
- The number of constraints is $|\mathcal{U}| + 1$.

There are other slightly different forms of linear programs to test dominated vectors, but asymptotically, the number of variables is always $O(|S|)$, and the number of constraints is always $O(|\mathcal{U}|)$.

Let $\mathbb{PR}(\mathcal{U})$ denote the operator that uses the above tests to prune vector set $\mathcal{U}$ to its minimal form. The minimum-size sets of vectors defined earlier can be computed as follows:

$$
\begin{aligned}
\mathcal{V}^{a,z} &= \mathbb{PR}\{v^{a,z,i} | v^i \in \mathcal{V}\}; \\
\mathcal{V}^{a} &= \mathbb{PR}\left\{\oplus_{z \in Z} \mathcal{V}^{a,z}\right\}; \\
\mathcal{V}' &= \mathbb{PR}\left\{\cup_{a \in A} \mathcal{V}^{a}\right\}.
\end{aligned}
$$

It is well known that solving linear programs in the pruning procedure $\mathbb{PR}$ takes up most of the computational time in the DP update [25]. Pruning algorithm plays a central role in many POMDP algorithms and is one of the focuses of this thesis.

## 2.7 Chapter notes

Thorough mathematical treatments on MDPs can be found in many references, such as [6, 62, 87]. The standard dynamic programming algorithms for MDPs date back at least to the 1950's [6]. The study of partially observable MDPs starts in the 1960's [39], and complete DP algorithms were first developed in the 1970's [95, 92, 96].

# CHAPTER 3

# SYMBOLIC HEURISTIC SEARCH FOR DISCRETE STATE MDPS

This chapter describes two new algorithms for solving large scale discrete state MDPs. They are based on a heuristic search approach toward solving MDPs. Traditional heuristic search algorithms for MDPs all rely on a flat state representation. The two algorithms in this chapter apply the symbolic representation used in the SPUDD algorithm (reviewed in Section 2.5.2) to heuristic search algorithms. In particular, the symbolic LAO* algorithm [44] represents currently one of the best optimal algorithms for solving large scale symbolic MDPs, and the symbolic RTDP algorithm [46] provides a novel framework for experience generalization in stochastic on-line planning. Although this chapter does not present new representations, it demonstrates the effectiveness a good representation can have on the efficiency of solution algorithms. Therefore this chapter serves as a motivation to the study of more advanced representations for increasingly more complex problems in later chapters.

## 3.1 Symbolic reachability analysis

While the SPUDD algorithm exploits state abstraction through the use of the ADD representation, it still solves the MDP for all states. In practice, if the starting state is known, then some states in the state space may not be reachable. Therefore the value of those states are irrelevant to making decisions and need not be computed at all. Many algorithms take advantage of this fact. For example, the LAO* algorithm [55], the RTDP algorithm [5], the envelope algorithm [35], and many reinforcement learning algorithms [98]. These algorithms are very similar to classical heuristic search algorithms. They are sometimes referred to as search algorithms for MDPs. The search component in these algorithms makes it possible to focus computations on reachable and relevant states. On the other hand, like classical search, these algorithms still rely on a flat representation of the state space which is subject to the state explosion problem. In contrast, state space traversal algorithms extensively used in the field of symbolic model checking operates directly using ADDs and BDDs and can alleviate many of the state explosion problems.

In symbolic model checking, a set of states $S$ is represented by its characteristic function $\chi_S$, where

$$\chi_S(s) = \begin{cases} 1 & \text{if } s \in S; \\ 0 & \text{if } s \notin S. \end{cases}$$

Since the characteristic function is a mapping from state variables to 0 and 1, it can be represented by an ADD. The traversal is computed by a series of *image* operations. Given a set of states represented by an ADD characteristic function, the image operator computes the set of all possible successor states under some transition function. To perform this operation, it is convenient to convert the transition ADD $P^a(\mathbf{X}'|\mathbf{X})$ to a 0-1 ADD $T^a(\mathbf{X}'|\mathbf{X})$ where

$$T^a(X_1', \ldots, X_n'|X_1, \ldots, X_n) = \begin{cases} 1 & \text{if } P^a(X_1', \ldots, X_n'|X_1, \ldots, X_n) > 0 \\ 0 & \text{otherwise} \end{cases}$$

The image operator can then be formally defined as

$$Image_{\mathbf{X}'}(S, T^a) = \exists \mathrm{x} \left[ T^a(\mathbf{X}, X') \wedge \chi_S(\mathbf{X}) \right].$$

Here the conjunction $T^a(\mathbf{X}, X') \wedge \chi_S(\mathbf{X})$ selects the set of valid transitions and the *existential quantification* $\exists \mathrm{x}$ extracts and unions the successor states together. These operations are well studied in the symbolic model checking literature, and there are efficient algorithms and implementations available [94]. The *Image* operator returns a characteristic

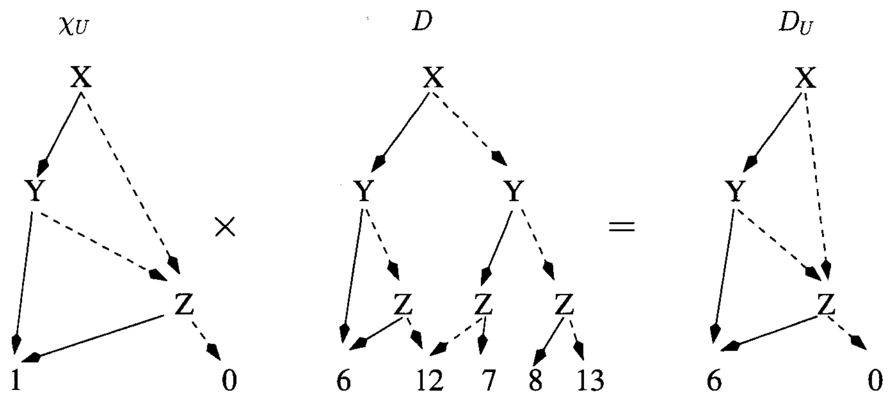


**Figure 3.1.** Masking a function $D$ with $\chi_U$, the characteristic function of a set of states $U$.

function over $\mathbf{X}'$ that represents the set of reachable states *after* an action is taken. A *swap* operator can be used switches variables in $\mathbf{X}'$ in an ADD with those in $\mathbf{X}$, effectively changing the pre-action and post-action status of a function. We will use the image operator to perform symbolic reachability analysis in off-line search (Section 3.2), and generalization in on-line search (Section 3.3).

## 3.2 Off-line heuristic search

Symbolic LAO* is an extension to LAO* [55] that uses ADDs to symbolically represent an MDP. In addition, it uses symbolic reachability analysis to find the set of relevant states and to limit the dynamic programming update, using what I call a *masking* operation.

Figure 3.1 shows the result of a masking operation. Given an ADD $D$ and a set of relevant states $U$, masking is performed by multiplying $D$ by $\chi_U$. This has the effect of mapping all irrelevant states to the value zero. [1] Let $D_U$ denote the resulting *masked ADD*. Mapping all irrelevant states to zero can simplify the ADD considerably. If the set of

[1]In fact, one can choose to map irrelevant states to arbitrary values. This suggests a way to simplify a masked ADD further. After mapping irrelevant states to zero, the value of an irrelevant state can be changed to any other non-zero value whenever doing so further simplifies the ADD [61].

reachable states is small, the masked ADD often has dramatically fewer nodes. This in turn can dramatically improve the efficiency of computation using ADDs.

Symbolic LAO* does not maintain an explicit search graph. It is sufficient to keep track of the set of states that have been *expanded* so far, denoted $G$, the *partial value function*, denoted $V_G$, and a *partial policy*, denoted $\pi_G$. For any state in $G$, we can "query" the policy to determine its associated action, and compute its successor states. Thus, the graph structure is implicit in this representation. Throughout the whole LAO* algorithm, only one value function $V$ and one policy $\pi$ is maintained. $V_G$ and $\pi_G$ are implicitly defined by $G$ and the masking operation. Symbolic LAO* is summarized in Table 3.1. The details are explained in the following sections.

### 3.2.1 Policy expansion

In the policy expansion step of the algorithm, reachability analysis is performed to find the set of states $F$ that are not in $G$ (i.e., have not been expanded yet), but are reachable from the set of start states, $S^0$, by following the partial policy $\pi_G$. These states are on the *fringe* of the states visited by the best policy. We add them to $G$ and to the set of states $E \subseteq G$ that are visited by the current partial policy. This is analogous to expanding states on the frontier of a search graph in heuristic search. Expanding a partial policy means that it will be defined for a larger set of states in the dynamic programming step.

Because a policy is associated with a set of transition functions, one for each action, we need to invoke the appropriate transition function for each action when computing successor states under a policy. For this, it is useful to represent the partial policy $\pi_G$ in another way. For each action $a$, there is a set of states in which $a$ is the action to be taken under the current policy. Call this set of states $S^a_\pi$. Note that $S^a_\pi \cap S^{a'}_\pi = \emptyset$ for $a \neq a'$, and $\cup_a S^a_\pi = G$. Given this alternative representation of the policy, line 4 computes the set of successor states following the current policy by applying the *Image* operator on each set $S^a_\pi$ and the transition function of $a$.

```
policyExpansion($\pi, S^0, G$)
1.   $E = F = \emptyset$
2.   $from = S^0$
3.   REPEAT
4.     $to = \bigcup_a Image(from \cap S^a_\pi, P^a)$
5.     $F = F \cup (to - G)$
6.     $E = E \cup from$
7.     $from = to \cap G - E$
8.   UNTIL ($from = \emptyset$)
9.   $E = E \cup F$
10.  $G = G \cup F$
11.  RETURN ($E, F, G$)

valueIteration($E, V$)
12.  $saveV = V$
13.  $E' = \bigcup_a Image(E, P^a)$
14.  REPEAT
15.    $V' = V$
16.    FOR each action $a$
17.        $V^a = R^a_E + \gamma \sum_{E'} P^a_{E \cup E'} V'_{E'}$
18.    $M = \max_a V^a$
19.    $V = M_E + saveV_{\overline{E}}$
20.    residual $= \|V_E - V'_E\|$
21.  UNTIL stopping criterion met
22.  $\pi = extractPolicy(M, \{V^a\})$
23.  RETURN ($V, \pi, residual$)

LAO*($\{P^a\}, \{R^a\}, \gamma, S^0, h, threshold$)
24.  $V = h$
25.  $G = \emptyset$
26.  $\pi = 0$
27.  REPEAT
28.      $(E, F, G) = policyExpansion(\pi, S^0, G)$
29.      $(V, \pi, residual) = valueIteration(E, V)$
30.  UNTIL ($F = \emptyset$) AND (residual $\leq$ threshold)
31.  RETURN ($\pi, V, E, G$).
```

**Table 3.1.** Symbolic LAO* algorithm.

### 3.2.2 Dynamic programming

The dynamic-programming step of LAO* is performed using a modified version of the SPUDD algorithm. The original SPUDD algorithm performs dynamic programming over the entire state space. We modify it to focus computation on reachable states, using the idea of masking. Masking lets us perform dynamic programming on a subset of the state space instead of the entire state space. The pseudo-code in Table 3.1 assumes that dynamic programming is performed on $E$, the states visited by the currently best (partial) policy. This has been shown to lead to the best performance of LAO*, although a larger or smaller set of states can also be updated [55]. Note that all ADDs used in the dynamic-programming computation are masked to improve efficiency.

Because $\pi_G$ is a partial policy, there can be states in $E$ with successor states that are not in $G$, denoted $E'$. This is true until LAO* converges. In line 13, we identify these states so that we can do appropriate masking. To perform dynamic programming on the states in $E$, we assign admissible values to the "fringe" states in $E'$, where these values come from the current value function. Note that the value function is initialized to an admissible heuristic evaluation function at the beginning of the algorithm.

With all components properly masked, we can perform dynamic programming using the SPUDD algorithm. This is summarized in line 17. The full equation is

$$V^a(\mathbf{X}) = R^a_E(\mathbf{X}) + \gamma \sum_{E'} P^a_{E \cup E'}(\mathbf{X}'|\mathbf{X}) \cdot V'_{E'}(\mathbf{X}'). \tag{3.1}$$

The masked ADDs $R^a_E$ and $P^a_{E \cup E'}$ need to be computed only once for each call to procedure valueIteration since they don't change between iterations. Note that the product $P^a_{E \cup E'} \cdot V'_{E'}$ is effectively defined over $E \cup E'$. After the summation over $E'$, which is accomplished by existentially abstracting away all post-action variables, the resulting ADD is effectively defined over $E$ only. As a result, $V^a$ is effectively a masked ADD over $E$, and the maximum $M$ at line 18 is also a masked ADD over $E$. In line 19, we use the notation $M_E$ to emphasize that $V$ is set equal to the newly computed values for $E$ and the saved values for the rest of the state space. There is no masking computation performed.

The residual in line 20 can be computed by finding the largest absolute value of the ADD $(V_E - V'_E)$. We use the masking subscript here to emphasize that the residual is computed only for states in the set $E$. The masking operation can actually be avoided here since at this step, $V_E = M$, which is computed in line 18, and $V'_E$ is the $M$ from the previous iteration.

Dynamic programming is the most expensive step of LAO*, and it is usually not efficient to run it until convergence each time this step is performed. Often a single iteration gives the best performance. After performing value iteration, we extract a policy in line 22 by comparing $M$ against the action value function $V^a$ (breaking ties arbitrarily):

$$\forall s \in E \;\; \pi(s) = a \text{ if } M(s) = V^a(s).$$

The symbolic LAO* algorithm returns a value function $V$ and a policy $\pi$, together with the set of states $E$ that are visited by the policy, and the set of states $G$ that have been "expanded" by LAO*.

### 3.2.3 Admissible heuristics

LAO* uses an admissible heuristic to guide the search. Because a heuristic is typically defined for all states, a simple way to create an admissible heuristic is to use dynamic programming to create an approximate value function. Given an error bound on the approximation, the value function can be converted to an admissible heuristic. (Another way to ensure admissibility is to perform value iteration on an initial value function that is admissible, since each step of value iteration preserves admissibility.) Symbolic dynamic programming can be used to compute an approximate value function efficiently. St. Aubin et al. [97] describe an approximate dynamic programming algorithm for factored MDPs, called APRICODD, that is based on SPUDD. It simplifies the value function ADD by aggregating states with similar values. Another approach to approximate dynamic programming for factored MDPs described by Dearden and Boutilier [37] can also be used to compute admissible heuristics.

Use of dynamic programming to compute an admissible heuristic points to a two-fold approach to solving factored MDPs. First, dynamic programming is used to compute an approximate solution for all states that serves as a heuristic. Then heuristic search is used to find an exact solution for a subset of reachable states.

### 3.2.4 Convergence

At the beginning of LAO*, the value function $V$ is initialized to the admissible heuristic $h$ that overestimates the optimal value function. Each time value iteration is performed, it starts with the current values of $V$. Hansen and Zilberstein (2001) show that these values decrease monotonically in the course of the algorithm; are always admissible; and converge arbitrarily close to optimal. LAO* converges to an optimal or $\epsilon$-optimal policy when two conditions are met: (1) its current policy does not have any unexpanded states, and (2) the error bound of the policy is less than some predetermined threshold. Like other heuristic search algorithms, LAO* can find an optimal solution without visiting the entire state space. The convergence proofs for the original LAO* algorithm carries over directly to symbolic LAO*.

### 3.2.5 Empirical evaluation

This section presents empirical data on the performance of symbolic LAO*. Note that the effectiveness of symbolic LAO* was also verified in the Probabilistic Track of the 2004 International Planning Competition, in which the algorithm won a first place award [73].

Table 3.2 compares the performance of LAO* and SPUDD on the factory examples (f to f6) used in [59] to test the performance of SPUDD, as well as some additional examples (a1 to a4). We use additional test examples because many of the state variables in the factory examples represent resources that cannot be affected by any action. As a result, we found

| Example | | | Reachability Results | | | Size Results | | | | Timing Results | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Symb-LAO* | | | Symb-LAO* | | SPUDD | | Symb-LAO* | | | LAO* | SPUDD |
| | $\|S\|$ | $\|A\|$ | $\|E\|$ | $\|G\|$ | reach | N | L | N | L | exp. | DP | total | total | total |
| f | $2^{17}$ | 14 | 5 | 105 | 190 | 55 | 5 | 1220 | 246 | 0.3 | 6.4 | 6.7 | 3.8 | 34.5 |
| f0 | $2^{19}$ | 14 | 5 | 62 | 132 | 61 | 5 | 1597 | 246 | 0.2 | 3.5 | 3.7 | 3.7 | 46.2 |
| f1 | $2^{21}$ | 14 | 4 | 54 | 107 | 54 | 4 | 3101 | 327 | 0.1 | 3.6 | 3.7 | 3.7 | 101.6 |
| f2 | $2^{22}$ | 14 | 4 | 66 | 125 | 53 | 4 | 3101 | 327 | 0.2 | 4.1 | 4.4 | 4.1 | 105.0 |
| f3 | $2^{25}$ | 15 | 4 | 59 | 136 | 74 | 4 | 9215 | 357 | 0.2 | 4.7 | 4.9 | 3.7 | 289.1 |
| f4 | $2^{28}$ | 15 | 4 | 49 | 125 | 78 | 4 | 22170 | 527 | 0.1 | 5.0 | 5.2 | 3.7 | 645.3 |
| f5 | $2^{31}$ | 18 | 5 | 218 | 509 | 83 | 4 | 44869 | 1515 | 1.2 | 35.9 | 37.3 | 4.4 | 1524.2 |
| f6 | $2^{35}$ | 23 | 9 | 1419 | 2386 | 106 | 5 | 169207 | 3992 | 13.5 | 771.5 | 792.6 | 9.3 | 7479.5 |
| a1 | $2^{20}$ | 25 | $3.0\times10^3$ | $3.3\times10^3$ | $1.0\times10^6$ | 181 | 19 | 15758 | 4056 | 0.5 | 53.6 | 54.0 | 39.01 | 12774.3 |
| a2 | $2^{20}$ | 30 | $1.0\times10^4$ | $3.3\times10^4$ | $1.0\times10^6$ | 6240 | 2190 | 9902 | 4594 | 57.7 | 1678.5 | 1738.1 | 4581.9 | 10891.7 |
| a3 | $2^{30}$ | 10 | $1.8\times10^6$ | $1.9\times10^6$ | $6.7\times10^7$ | 3522 | 439 | 25839 | 6434 | 7.3 | 344.8 | 352.1 | N/A | 11169.9 |
| a4 | $2^{40}$ | 10 | $9.6\times10^4$ | $2.8\times10^6$ | $1.7\times10^{10}$ | 99 | 4 | N/A | N/A | 0.7 | 75.5 | 76.3 | N/A | N/A |

**Table 3.2.** Performance comparison of LAO* (both symbolic and non-symbolic) and SPUDD.

that only a small number of states are reachable from a given start state in these examples. Examples a1 to a4 are modified version of the factory examples where every state variable can be changed by some action, and thus, most or all of the state space can be reached from any start state. Such examples present a greater challenge to a heuristic-search approach.

Because the performance of LAO* depends on the start state, the experimental results reported for LAO* in Table 3.2 are averages for 50 random starting states. To create an admissible heuristic, we performed several iterations (ten for the factory examples and twenty for the others) of an approximate value iteration algorithm similar to APRICODD [97]. The algorithm was started with an admissible value function created by assuming the maximum reward is received each step. The time used to compute the heuristic for these examples is between 2% and 8% of the running time of SPUDD on the same examples.

LAO* achieves its efficiency by focusing computation on a subset of reachable states. The column labeled $|E|$ shows the average number of states visited by an optimal policy, beginning from a random start state. Clearly, the factory examples have an unusual structure, since an optimal policy for these examples visits very few states. The numbers are much larger for examples a1 through a4. The column labeled *reach* shows the average number of states that can be reached from the start state, by following *any* policy. The column labeled $|G|$ is important because it shows the number of states "expanded" by LAO*. These are states for which a backup is performed at some point in the algorithm, and this number depends on the quality of the heuristic. The better the heuristic, the fewer states need to be expanded to find an optimal policy. The gap between $|E|$ and *reach* reflects the potential for increased efficiency using heuristic search, instead of simple reachability analysis. The gap between $|G|$ and *reach* reflects the actual increased efficiency.

The columns labeled N and L, under LAO* and SPUDD respectively, compare the size of the final value function returned by symbolic LAO* and SPUDD. Columns N give the number of nodes in the respective value function ADDs. Columns L give the number of leaves. Because LAO* focuses computation on a subset of the state space, it finds a much more compact solution (which translates into increased efficiency).

The last five columns compare the average running times of symbolic LAO* to the running times of non-symbolic LAO* and SPUDD. Times are given in CPU seconds. For many of these examples, the MDP model is too large to represent explicitly. Therefore, our implementation of non-symbolic LAO* uses the same ADD representation of the MDP model as symbolic LAO* and SPUDD. However, non-symbolic LAO* performs heuristic search in the conventional way by creating a search graph in which the nodes correspond to "flat" states that are enumerated individually.

The total running time of symbolic LAO* is broken down into two parts; the column "exp." shows the average time for policy expansion and the column "DP" shows the average time for dynamic programming. These results show that dynamic programming consumes most of the running time. This is in keeping with a similar observation about the original (non-symbolic) LAO* algorithm. The time reported for dynamic programming includes the time for masking. For this set of examples, masking takes between 0.5% and 2.1% of the running time of the dynamic programming step. The last three columns compare the average time it takes symbolic LAO* to solve each problem, for 50 random starting states, to the running times of non-symbolic LAO* and SPUDD. This comparison leads to several observations.

First, note that the running time of non-symbolic LAO* is correlated with $|G|$, the number of states expanded and evaluated during the search, which in turn is affected by the starting state, the reachability structure of the problem, and the accuracy of the heuristic function. As $|G|$ increases, the running time of non-symbolic LAO* increases. The search graphs for examples a3 and a4 are so large that these problems cannot be solved using non-symbolic LAO*. (N/A indicates that the problem could not be solved and no data is available.)

The running time of symbolic LAO* depends not only on $|G|$, but also on the degree of state abstraction the symbolic approach achieves in representing the states in $G$. For the factory examples and example a1, the number of states evaluated by LAO* is small enough that the overhead of symbolic search outweighs the improved efficiency from state abstraction. For these examples, symbolic LAO* is somewhat slower than non-symbolic LAO*. But for examples a2 to a4, the symbolic approach significantly – and sometimes dramatically – improves the performance of LAO*. Symbolic LAO* also outperforms SPUDD for all examples. This is to be expected since LAO* solves the problem for only part of the state space. Nevertheless, it demonstrates the power of using heuristic search to focus computation on relevant states.

Note also that examples a3 and a4 are beyond the range of both SPUDD and non-symbolic LAO*, or can only be solved with great difficulty. Yet symbolic LAO* solves both examples efficiently. This illustrates the advantage of combining heuristic search and state abstraction, rather than relying on either approach alone.

### 3.2.6 Summary

We have described a symbolic generalization of LAO* that solves factored MDPs using heuristic search. Given a start state, LAO* uses an admissible heuristic to focus computation on the parts of the state space that are reachable from the start state. The stronger the heuristic, the greater the focus and the more efficient a planner based on this approach. Symbolic LAO* also exploits state abstraction using symbolic model checking techniques. It can be viewed as a decision-theoretic generalization of symbolic approaches to nondeterministic planning.

## 3.3 On-line generalization

Since LAO* and symbolic LAO* are off-line algorithms, the agent must wait till the full optimal solution is found before acting. For problems that requires faster response time, it is desirable to act sooner at the cost of sub-optimal performance. Real-time dynamic programming (RTDP) [5] is an algorithm that offers such trade-off. In RTDP, the search and dynamic programming happens in real-time and are more closely integrated. Because of this, RTDP usually achieves a good value function faster than LAO*, although eventually it takes longer than LAO* to converge to the optimal [55].

The basic RTDP algorithm works as follows. At each time step $t$, the agent observes the current state $s_t$ and immediately performs a DP backup to update its value:

$$V^{t+1}(s_t) \leftarrow \max_{a \in A} \left\{ R^a(s_t) + \gamma \sum_{s' \in S} P^a(s_t, s') V^t(s') \right\}.$$

The values of all other states are kept unchanged, that is, for all $s \neq s_t$:

$$V^{t+1}(s) = V^t(s).$$

If the initial value function is an admissible estimate of the optimal value function, then an agent can always take the action that maximizes Equation 3.3. Otherwise some exploration scheme must be used in choosing actions, in order to ensure convergence. After an action is taken, the agent observes the resulting state and the above process repeats.

As mentioned before, although RTDP provides good initial results relatively quickly, it can take a long time to converge. This is due to the enumerative nature of the trajectory sampling. When the state space is large enough, a state by state update becomes very inefficient, especially if the sampling in the domain involves carrying out physical actions. Therefore some way of generalizing the limited experience is necessary. The symbolic RTDP algorithm presented in this section uses symbolic model checking techniques to generalize experience from a single state to a group states, and perform DP update on the resulting group.

There are many ways to group states into abstract states. In this section, two heuristic approaches are studied that are motivated by the idea of generalization by structural similarity. A *value-based* generalization creates an abstract state consists of states whose value estimates are close to that of the current state. A *reachability-based* generalization creates an abstract state that consists of states that share with the current state a similar set of successor states. Unlike SPUDD, we *explicitly* construct these abstract states at each time step of symbolic RTDP, using standard ADD model checking operators.

### 3.3.1 Generalization by value

With a value-based abstract state, the experience is generalized to states that have similar value estimates as the current state. The intuition is that states with similar *optimal* values may also be similarly desirable. Generalizing updates to states with similar *estimated* values helps the agent in two ways. First, if some of these states indeed have similar optimal value as the current state, the update strengthens this similarity and the agent is better informed in the future when these states are visited again. Second, if some of the states have very different optimal value than the current state, the generalization helps to distinguish them and avoid computations on them in the future when the same state as the current state is visited again.

Let $s$ be the current state and $V$ be the current value function. The characteristic function of the value-based abstract state $E$ can be constructed by setting leaf nodes in $V$ with values close to $V(s)$ to 1, and all other leaf nodes to 0. The change at the leaf nodes then propagates up to the root. This operation is standard in most ADD packages, including CUDD [93], the one we use for our implementation. The complexity of this operation is $O(|V|)$, where $|V|$ is the number of nodes in the ADD representation of $V$.

### 3.3.2 Generalization by reachability

With a reachability-based abstract state, the experience is generalized to states that are similar to the current state in terms of the set of one-step reachable states. The intuition here is that if the agent is going to visit some states, say $C$, from the current state $s$, then any information about $C$ is useful not only to $s$ but also to other states that can reach $C$.

By generalizing the update to these other states the agent is better informed in the future whether to aim at $C$ or to avoid it.

To compute the abstract state based on reachability, we introduce a new operator from the model checking literature. Similar to the $Image$ operator (Section 3.1) which computes a set of reachable states, the $PreImage(C)$ operator computes the set of states that can reach some state in $C$ in one step. The reachability-based abstract state $E$ can then be computed as:

$$E = PreImage(Image(\{s\})) - PreImage(S - Image(\{s\})).$$

### 3.3.3 Symbolic RTDP

Table 3.3 shows the pseudo-code of a trial-based version of symbolic RTDP. It takes as input an admissible initial value function $V_0$, a starting state $s_0$, the number of trials to run, and the number of steps to run in each trial. It returns an updated value function, from which a policy can be extracted.

We extend the idea of masking in symbolic LAO* to symbolic RTDP by performing DP on the abstract state $E$ that the current state $s$ belongs to. The abstract state is constructed by one of the generalization procedures mentioned in previous sections. In the pseudo-code, this is denoted by the function $Generalize(s)$.

Once the set $E$ is computed, it is used to mask the current value function before perform the DP update, using the same masking process described for symbolic LAO* (Equation 3.1). After the update, an action is chosen that maximizes the DP update at state $s$. The agent then carries out the action, and the process repeats.

Although both symbolic LAO* and symbolic RTDP use a "masked" DP update, the masks they use are different and serve different purposes. The mask in symbolic LAO* contains all states visited so far by the forward search step. The purpose of masking is to restrict computation to relevant states. The mask in symbolic RTDP contains states that share structural similarity with the state being visited currently. The purpose of masking

```
**SymbolicRTDP**($V_0, s_0, nTrials, nSteps$)
1.  $V \leftarrow V_0$;
2.  Repeat $nTrials$ times
3.    $s \leftarrow s_0$;
4.    Repeat $nSteps$ times
5.      $E \leftarrow Generalize(s)$
6.      $E' \leftarrow$ States reachable in one step from $E$
7.      $V^{copy} \leftarrow V$
8.      For all $a \in A$:
9.        $Q_a \leftarrow R^a_E(\mathbf{X}) + \gamma \exists_{E'} P^a_{E \cup E'}(\mathbf{X}, \mathbf{X}') \cdot V_{E'}(\mathbf{X}')$
10.     $V_E \leftarrow \max_a Q_a$
11.     $a \leftarrow \arg\max_a Q_a(s)$
12.     $V \leftarrow V_E + V^{copy}_{\bar{E}}$
13.     $s \leftarrow Execute(s, a)$
14. Return $V$
```

**Table 3.3.** Trial-based symbolic RTDP algorithm

here is to generalize update on a single state to an abstract state. This generalization has two consequences. It introduces some overhead in the DP step, including identifying the abstract state, and performing masked DP instead of single-state DP. On the other hand, it updates the value of a group of states in a single step, at a cost that can be significantly less than updating the states separately, because the symbolic computation exploits state abstraction.

### 3.3.4 Adaptive symbolic RTDP

Barto *et.al* [5] describe an adaptive version of RTDP where the model parameters are not known and have to be estimated on-line while the agent is acting. It is straightforward to extend symbolic RTDP to this setting. We call this algorithm adaptive symbolic RTDP. Learning algorithms developed for Bayesian networks can be applied to learn the model parameters of a factored MDP, for example [50, 91]. Since model learning is not the focus of this thesis, we introduce two assumptions for this task to simplify implementation in the empirical test: 1) The reward function is given; and 2) The structure of the transition

ADD $P^a(\mathbf{X}'|\mathbf{X})$ is given for all actions $a$. In other words, only the probabilities need to be estimated. Note that similar work in prioritized sweeping [36, 1] use the same assumption. Given these assumptions, it is straightforward to design a maximum-likelihood algorithm to estimate the missing probabilities.

With model parameters updated properly at each step, we need to modify the symbolic RTDP algorithm by using the learned model in Equation 3.1. The identification of the abstract state remains the same. To satisfy the convergence conditions for adaptive RTDP, we use a simple $\epsilon$-greedy exploration schemes [98] to replace the action selection step at line 11 of the algorithm. Finally, since there is no model to begin with, it is generally not possible to compute admissible heuristic systematically. But a good estimate can still speed up convergence.

### 3.3.5 Convergence

If we implement the function $Generalize(s)$ so that it only returns $\{s\}$, then symbolic RTDP becomes the original RTDP. On a state by state level, the only difference between RTDP and symbolic RTDP is that RTDP update the current state only, while symbolic RTDP update the current state *and* some other states.[2] Thus, if the convergence conditions for RTDP are met, symbolic RTDP will also converge.

**Theorem 1** *Symbolic RTDP converges to the optimal value function under the same conditions that RTDP converges if for every state s, $s \in Generalizes(s)$.*

**Proof** The convergence proof of the original RTDP relies on converting the sequential update of states by RTDP to an asynchronous dynamic programming process [5]. It has been shown that under various conditions, this asynchronous DP process converges to the optimal value function, as long as the current state is always updated. Symbolic RTDP

[2]Note that this possibility of updating extra states other than the current state is mentioned in the original RTDP paper [5]. The symbolic generalization technique presented here provides a systematic and efficient way of choosing the extra states and updating their values.

can be analyzed in exactly the same way by converting it to an asynchronous DP process. And since $s \in Generalizes(s)$ hold for all $s$, the current state is always updated, in addition to some other states. Updating these other states only improves their value, thus the asynchronous DP process will still converge, if only faster. ■

### 3.3.6 Empirical evaluation

In this section, we consider the empirical performance of symbolic RTDP and adaptive symbolic RTDP, and the generalization behavior of our two techniques for identifying an abstract state. We compare the performance of symbolic RTDP to both RTDP and symbolic LAO*, and compare the performance of adaptive symbolic RTDP to an adaptive version of RTDP. In our comparison, all algorithms use the same symbolic representation of the problem. The reason the non-symbolic algorithms must use a symbolic representation is that our test problems are so large that a traditional table-based representation of the transition matrix cannot fit in memory. However, non-symbolic RTDP performed single-state DP backups using Equation 3.3 in our comparison, and does not exploit the symbolic representation in solving the MDP.

We tested the various algorithms on the same test problems used in Section 3.2.5, especially the most difficult of these problems, numbered *a1* through *a4*. The results for these four problems were similar and we only report results for problem *a1* here. For all algorithms, we use the same starting states. For all the non-adaptive versions of RTDP, we use the same admissible heuristic function, and for the adaptive version of RTDP, we initialize the value function to zero.

#### 3.3.6.1 Symbolic RTDP

In our experiments, the on-line or RTDP planning algorithms performed 100 trials, each consisting of 20 steps from the starting state. The off-line planner, symbolic LAO*, ran until convergence.

Figure 3.2 compares the performance of these algorithms. The $x$-axis shows the CPU time measured in seconds. The $y$-axis shows the value of the start state. Each point on the symbolic LAO* curve represents an iteration of forward search, followed by a backward DP update. Each point on the three RTDP curves represents a trial of 20 steps. As we can see, the two symbolic versions of RTDP perform much better than the original RTDP. This is because the symbolic approaches generalize experience and exploit state abstraction, while the original approach does not. Symbolic RTDP also compares favorably with symbolic LAO*. In particular, symbolic RTDP with generalization by value quickly reaches a near optimal value in the early stage of computation, while symbolic LAO* gradually catches up after about 100 seconds. Symbolic LAO* converges after running about 8 minutes, while symbolic RTDP continues without reaching the same value even at the end of the 100 trials. This behavior – in which symbolic RTDP improves a solution more quickly at first, and symbolic LAO* achieves eventual convergence faster – is similar to behavior observed in comparing non-symbolic versions of LAO* and RTDP [44].

From Figure 3.2, we can also see that symbolic RTDP takes longer to finish each trial than RTDP. In fact, RTDP finishes 100 trials in about 500 seconds, while the two symbolic RTDP algorithms only finish 20 to 40 trials at the same time. However, in each trial symbolic RTDP improves the value function more than RTDP. If we plot the curves of the three RTDP algorithms against the number of trials, shown in Figure 3.3, the difference is more obvious. After about 20 trials, symbolic RTDP reaches a value that is within 0.1 of the value that symbolic LAO* converges to. For RTDP, the difference in value is larger than 2.1 after 100 trials. Because RTDP updates a single state only at each step, it takes less time to finish a trial than symbolic RTDP, which performs the extra work of identifying and updating the abstract states at each step. However, the extra work performed by symbolic RTDP is more than justified by the improved performance it achieves due to state abstraction and generalization.

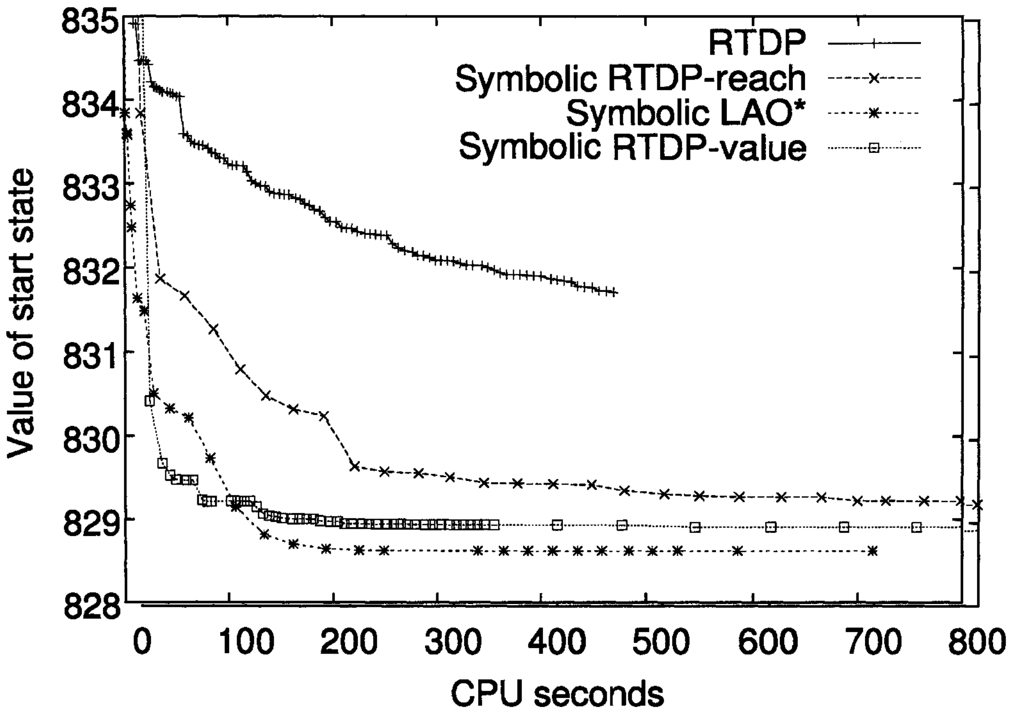


**Figure 3.2.** Performance comparison in CPU time

From Figures 3.2 and 3.3, we can see that the two notions of generalization work similarly well for this problem, with generalization by value slightly better than generalization by reachability. We expect that the relative performance of the two methods will depend on the characteristics of a problem. In particular, if the current value estimation is close to the underlying optimal value function, as is the case when an admissible heuristic is used, value based generalization should work better. Otherwise reachability based generalization can be more effective, as we will see in the next experiment.

### 3.3.6.2 Adaptive symbolic RTDP

We next compare adaptive versions of symbolic RTDP that uses the two generalization approaches, with an adaptive version of non-symbolic RTDP. The initial value function

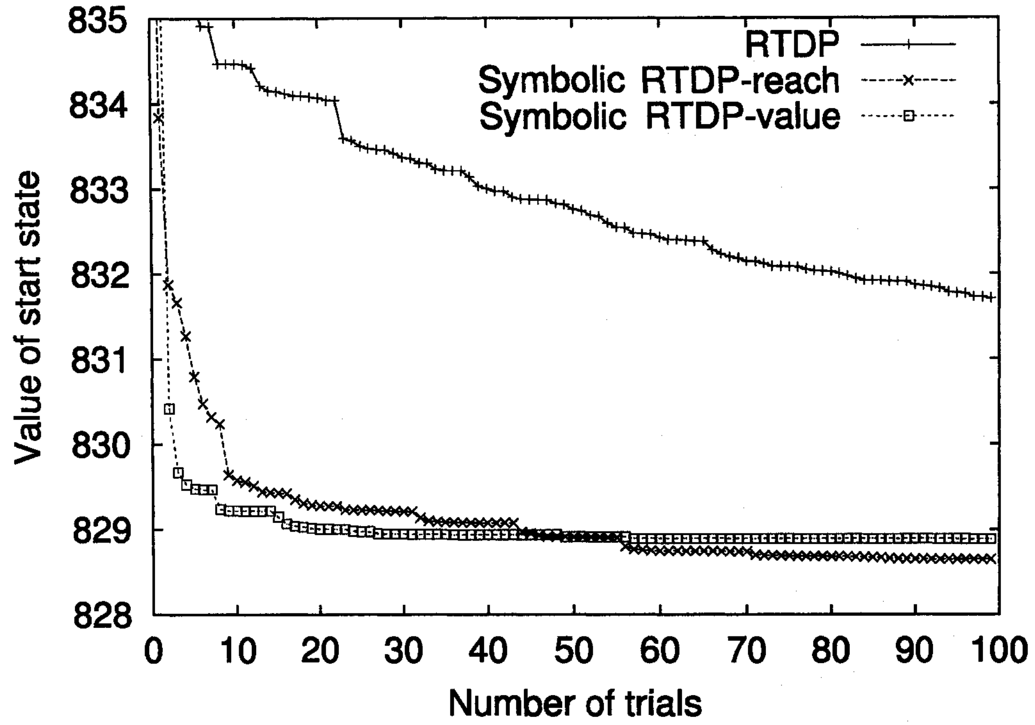


**Figure 3.3.** Performance comparison in number of trials

was set to 0 in our experiments. Figure 3.4 shows the results. Each curve represents the accumulated reward in each trial, and is averaged over 100 runs and smoothed. Each run contains 200 trials with 20 steps per trial.

As we can see, the two symbolic RTDP algorithms consistently outperform non-symbolic RTDP. Moreover, while we see a clear trend that the symbolic RTDP curves are improving, the non-symbolic RTDP curve seems to show no improvement over time. This is because symbolic RTDP generalizes its on-line experience, while RTDP does not. Recall that the problem has 20 state variables, or 1,048,576 states. Each run performs $200 \times 20 = 4,000$ times of sampling, which is less than 0.4% of the state space. (Considering some states may be sampled more than once, the actual sample coverage is likely to be smaller.) Since RTDP does not generalize, sample coverage at this magnitude is far from enough. symbolic

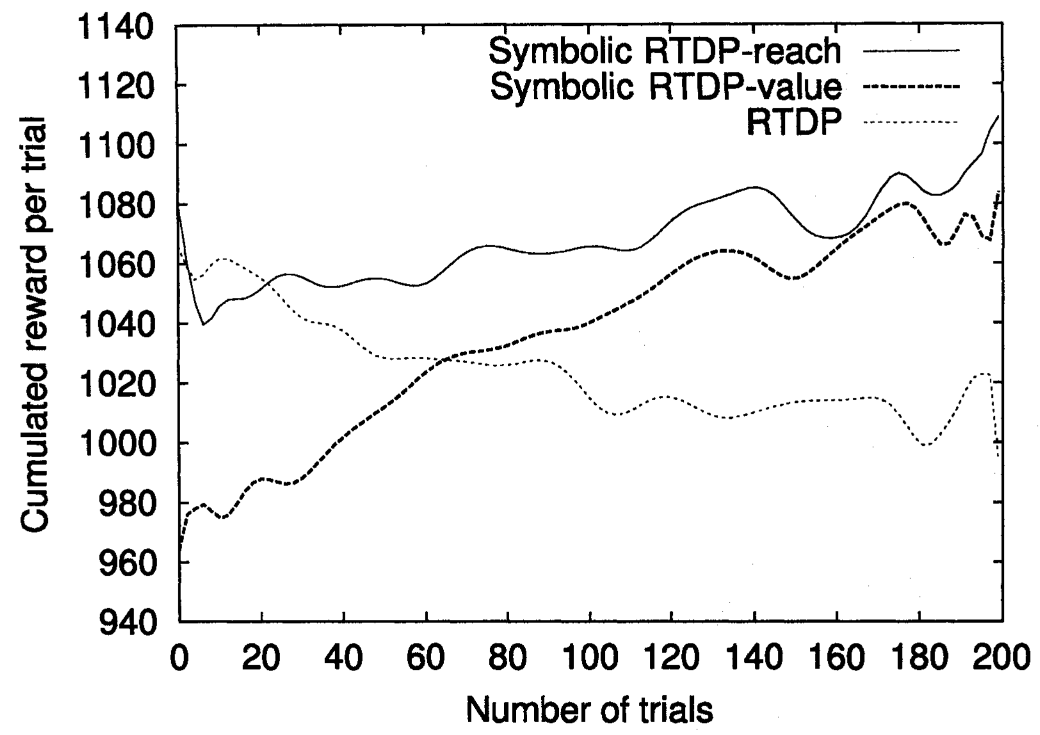


**Figure 3.4.** Performance comparison of adaptive symbolic RTDP and adaptive RTDP, averaged over 100 runs and smoothed.

RTDP, on the other hand, generalizes beyond the actual samples, and is able to improve its performance based on the same amount of experience that's available to RTDP.

By comparing the two symbolic RTDP curves, we can see that generalization by reachability performs better than generalization by value. In fact, generalization by value has the worst online performance among the three algorithms in the first 60 trials. This is because in the early stage, the value estimates are very inaccurate, so the computation performed by generalization by value is mainly geared toward distinguishing states that have similar estimates but indeed have different optimal values. As experience accumulates, the value estimates become more accurate and generalization by value can better exploit it to gather more reward.

We point out that, although the curves in Figure 3.4 look a bit noisy, the difference we discussed is statistically significant. We performed $t$-tests at various points of interest in the data to confirm this. In particular, around trial 20, there is significant difference between the two generalization methods. (The probability that the two are the same is 0.078.) In later trials, as the two curves cross, the significance of the difference reduces. Finally, at the end of the 200 trials, the difference between any two of the three algorithms is again statistically significant.

### 3.3.7 Discussion

Generalization has been long recognized as a crucial component of efficient planning and learning. It accelerates the learning process and reduces the amount of interaction with the environment needed to reach a desired level of competence. Many forms of generalization have been developed for deterministic and stochastic domains, but none has been proved effective for on-line planning in stochastic environments. We have described symbolic RTDP, an extension of RTDP that uses symbolic model checking techniques as an approach to generalizing experience in solving factored MDPs. By identifying and updating abstract states instead of single states, symbolic RTDP improves a state evaluation function faster than RTDP not only in terms of CPU time, but also in terms of the number of steps of interaction with the environment. This is particularly desirable when performing real-world actions is more expensive than performing computation, which is the case for many applications. The result is a novel generalization technique for on-line planning that accelerates convergence without compromising optimality. While generalization seems to always be beneficial, the effectiveness of different generalization methods may vary at different stages of on-line planning. Intuitively, generalizations that group together a larger set of situations are more powerful, but they are also more risky when they are based on little experience. It remains an open problem how to further accelerate convergence using mixed strategies that apply different forms of generalization at different stages of the interaction with the environment.

## 3.4 Related work

### 3.4.1 Related work in off-line planning

Symbolic model checking techniques have been used previously for nondeterministic planning. In both nondeterministic and decision-theoretic planning, plans may contain cycles that represent iterative, or "trial-and-error," strategies. In nondeterministic planning, the concept of a *strong cyclic plan* plays a central role [31, 33]. It refers to a plan that contains an iterative strategy and is guaranteed to eventually achieve the goal. The concept of a strong cyclic plan has an interesting analogy in decision-theoretic planning. LAO* was originally developed for the framework of stochastic shortest-path problems. A stochastic shortest-path problem is an MDP with a goal state, where the objective is to find an optimal policy (usually containing cycles) among policies that reach the goal state with probability one. A policy that reaches the goal with probability one, also called a *proper policy*, can be viewed as a probabilistic generalization of the concept of a strong cyclic plan. In this respect and others, the symbolic LAO* algorithm presented in this chapter can be viewed as a decision-theoretic generalization of symbolic algorithms for nondeterministic planning.

One difference is that the algorithm presented in this paper uses heuristic search to limit the number of states for which a policy is computed. An integration of symbolic model checking with heuristic search has not yet been explored for nondeterministic planning. However, Edelkamp describes a symbolic generalization of A* that combines symbolic model checking and heuristic search in solving deterministic planning problems [41]. A combined approach has also been explored for conformant planning [7].

In motivation, our work is closely related to the framework of *structured reachability analysis*, which exploits reachability analysis in solving factored MDPs [13]. However, there are important differences. The symbolic model-checking techniques we use differ from the approach to state abstraction used in that work, which is derived from GRAPH-PLAN [12]. More importantly, their concept of reachability analysis is weaker than the approach adopted here. In their framework, states are considered irrelevant if they cannot be reached from the start state by following *any policy*. By contrast, our approach considers states irrelevant if it can be proved (by gradually expanding a partial policy guided by an admissible heuristic) that these states cannot be reached from the start state by following *an optimal policy*. Use of an admissible heuristic to limit the search space is characteristic of heuristic search, in contrast to simple reachability analysis. As Table 3.2 shows, LAO* evaluates much less of the state space than simple reachability analysis. The better the heuristic, the smaller the number of states it examines.

Symbolic representation has been applied to classical planning as well, mainly using BDDs [41, 7, 64, 30]. These work mainly build on the space traversal techniques provided by symbolic model checking, and do not involve complex probabilistic reasonings. More expressive representations such as first-order logic have also been applied to model and solve MDPs with considerable success [18, 60].

### 3.4.2 Related work in on-line learning

In the original RTDP paper [5], it is suggested that updating extra states other than the current state at each step may improve the rate of convergence. On the other hand, using a flat representation, updating a large number of extra states will also slow down the whole algorithm. Further, it is not clear how to choose these extra states and how efficient such choice can be made. Symbolic RTDP provides a practical solution to all these issues.

Symbolic RTDP is closely related to Prioritized Sweeping (PS) [80, 1, 36], another class of algorithms that perform backups on multiple states for each episode of real-world interaction. In PS, each state is assigned a priority that is proportional to the change in the current value estimation. The algorithm then perform backups on states in order of their priority until the next action is taken. In particular, the Structured PS algorithm by Dearden [36] uses a decision tree data structure to update a group of states with the same priority using a *local decision-theoretic regression* operator, which is equivalent to the masked update operator used by symbolic RTDP and symbolic LAO*.

Symbolic RTDP is different from PS in the way it chooses extra states to be backed-up. Symbolic RTDP extends off-line planning algorithms that exploit problem structures, such as SPUDD and symbolic LAO*, to an on-line setting. It generalizes on-line experiences to states that are similar to the current state as measured by the similarity of the underlying value or reachability structure. When these structures are present in the problem, generalization improves on-line performance with very small computation overhead. PS, on the other hand, chooses extra states according to their priority, which is a measure of the changes in the value estimation. It is not clear whether the special value or reachability structure in a problem translates directly to exploitable structures in the space of priorities.

The idea of extending a backup of a single state to an abstract state is also closely related to function approximation methods for solving MDPs (See [21] for a brief review.) However, our work is fundamentally different in that we use an *exact* representation of the problem and the value function, while function approximation methods use an *approximate* representation. As a result, our approach preserves the guarantee of convergence and optimality of an algorithm, whereas most function approximation methods do not [21]. On the other hand, our representation does not exclude the possibility of approximation. By grouping similar but not identical state values together, we can reduce the size of the ADDs and the DP update can be computed more efficiently. This form of approximation has been studied in standard DP algorithms [97, 43] and shown to converge with bounded error. Those results can be extended to symbolic LAO* and symbolic RTDP as well.

Our work is also related to the idea of model minimization for MDPs, presented in [34]. Their algorithm constructs a *stochastic bisimulation* [68] for a symbolically represented MDP. The bisimulation consists of abstract states that are equivalent in terms of optimal value and optimal policy. A potentially smaller MDP can be constructed over this abstract state space and the optimal solution for it is also optimal for the original MDP. Our algorithm can be seen as an on-line version of model minimization, similar to [100], interleaved with an update of the value function using dynamic programming. Unlike MDP model minimization, we treat states as equivalent based on the current value estimate or local reachability structure, not the ultimate optimal value function. Thus, states that are "accidentally" equivalent in the short term are not distinguished from states that are truly equivalent in the long term. As a result, symbolic RTDP traverses an abstract state space that is potentially much smaller than the one that would be created by MDP model minimization.

# CHAPTER 4

# STATE ABSTRACTION FOR STRUCTURED CONTINUOUS MDPS

This chapter describes an important subclass of continuous state MDPs developed in collaboration with NASA researchers [42]. It models the control of a Mars rover with multiple continuous resources that need to be considered as part of the state space [23, 42]. The representation introduced here is inspired by the symbolic representation of discrete state MDPs. The state abstraction is created by partitioning the continuous space into rectangle regions. Each region is treated as an abstract state. Value functions over the state space are collectively defined by value functions over the regions. Since each region is still a continuous space, the piece-wise linear and convex value function representation used in standard POMDP algorithms (Section 2.6) is applied here to represent functions over the regions. This in turn relates to the region-based representation for POMDPs to be presented in Chapter 6.

## 4.1 Characteristics of the Mars rover domain

Consider a rover exploring Mars by carrying out various experiments. Different experiments have different scientific values associated with them. Each experiment may consume various amount of resources such as time, energy, data storage and/or communication bandwidth. For many of these experiments, there is inherent uncertainty about the amount of resources that may be consumed. For example, when moving between two locations, the amount of time required depends on the slope, roughness and soil characteristics of the terrain between the two locations, and the wheel slippage and sink-age [23]. It is impossible to model these factors precisely and therefore the time needed to move from one location to another can only be modeled probabilistically according to some statistic estimations of the various factors involved. The uncertainty about time also contributes to uncertainty about the amount of energy that will be used as well.

There are also constraints imposed on the experiments. For example, the action of driving requires that the energy is above certainty level to initiate the movement, although the level of energy needed to sustain a certain speed may be lower. This translates to an energy threshold above which driving can be effectively carried out. In another example, taking a picture may only be performed when the sun is at a certain angles to provide sufficient lighting. This translates to a time window during which experiments involving picture taking can be performed.

These constraints on actions and experiments essentially partition the time-energy space into rectangle regions. Within each region, the effects of an action are assumed to be the same regardless of the exact location at which that action is taken in that region. We will show that based on this assumption, the DP process can be carried out in a space consisting of rectangle regions in the continuous belief space.

Bresina [23] presented an example of such a domain, depicted in Figure 4.1. Each rectangle box represents an action. Arrows represents precedence constraints between actions. Each action consumes time and energy according to some Normal distributions, and some actions have resource constraints imposed (e.g. $E > 10Ah$ for Visual Servo).

In this example, there are potentially two experiments to be considered. The primary experiment involves approaching a target point (VisualServo), digging the soil (Dig), backing up a little bit (Drive), and taking several spectral images of the area (NIR). Taking spectral images requires considerable energy ($E > 3Ah$), therefore a backup plan is to take a high-resolution optical image (Hi res) instead, which only need a little energy ($E > .02Ah$). For this experiment, if the spectral images are successfully taken, then its scientific value is 100. If only optical images are taken, then the value is 10. A secondary experiment involves taking a low-resolution picture of the area (Lo res), invoking on-board image analysis routines

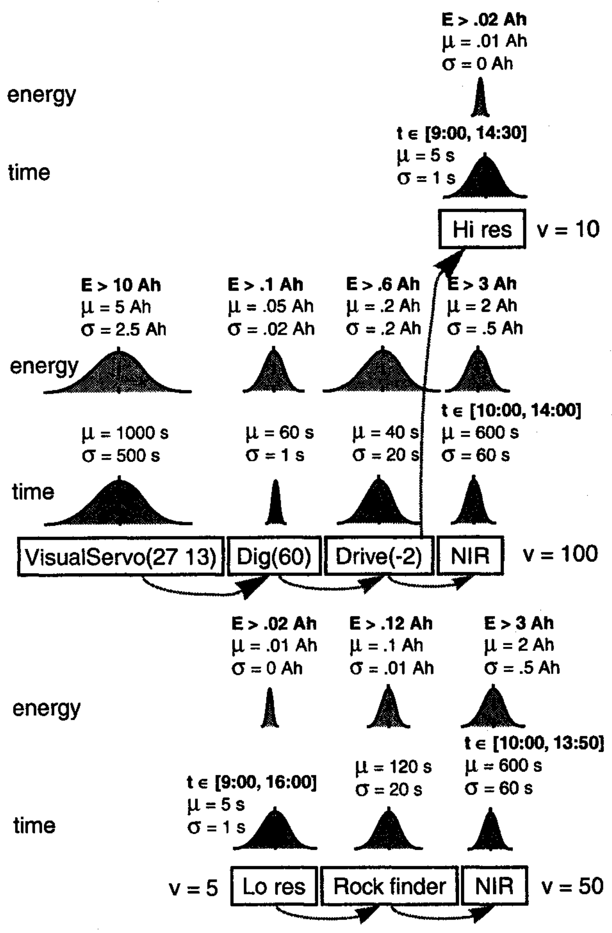


**Figure 4.1.** Example Mars rover planning domain (Source: [23])

to find rocks in the image (Rock finder), and then taking spectral images of the rocks found (NIR). The value of this experiment is 50.

Figure 4.2 shows the optimal value function of this problem over the continuous space of time and energy. The shape of this value function is characteristic of the rover domain, as well as other domains featuring a finite set of goals with positive utility and resource constraints. Such a value function features a set of humps and plateaus, each of them representing a region of the state space where a particular goal (or set of goals) can be reached. The sharpness of a hump or plateau reflects the uncertainty attached to the actions leading to this goal. Moreover, constraints on the minimal level of resource required to

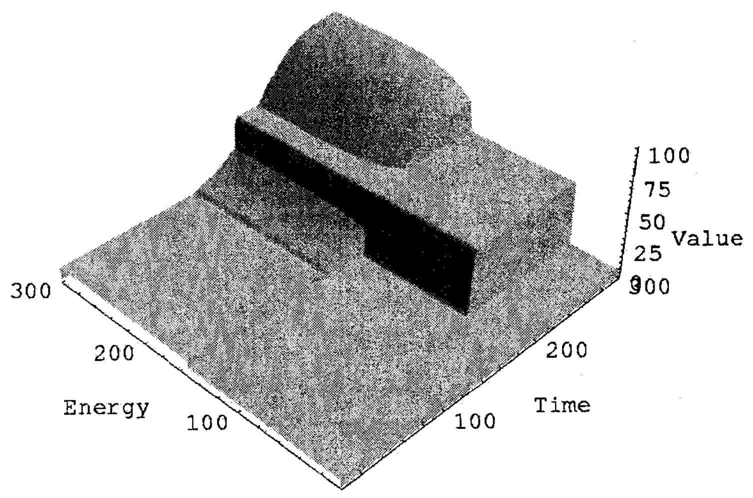


**Figure 4.2.** Value function of the Mars rover domain

start some actions introduce abrupt cuts in the regions. The goal of this chapter is to exploit such structure by grouping together states belonging to the same plateau, while reserving a fine discretization for the regions of the state space where it is the most useful (such as the curved hump where there is more time and energy available).

## 4.2 Basic model

Consider an MDP model with a continuous state space: $\{\mathbf{X}, A, P, R, \}$, where

- $\mathbf{X}$ is a vector of continuous state variables $\langle X_1, \ldots, X_d \rangle$. Without loss of generality, assume the value of the variables are all in the range $[0, 1)$, so the state space is the unit square $[0, 1)^d$. $\mathbf{x} \in [0, 1)^d$ refers to a particular state.

- $A$ is a finite set of actions.

- $P$ is the transition model. Following [20], both *relative* and *absolute* transitions are allowed. A relative transition is expressed as $P^a(\mathbf{x} + \delta\mathbf{x}, \mathbf{x})$, the probability that the state is shifted by $\delta\mathbf{x}$ relative to $\mathbf{x}$. An absolute transition is expressed as $P^a(\mathbf{x}', \mathbf{x})$, the probability that the resulting state is $\mathbf{x}'$. The finite set of possible resulting states from taking action $a$ in state $\mathbf{x}$ is referred to as the *outcomes* and denoted $\Delta^a_{\mathbf{x}}$. For relative outcomes, an element $\delta_i \in \Delta^a_{\mathbf{x}}$ is a pair $(\delta\mathbf{x}, p)$, where $p$ is the probability of that outcome. Similarly for absolute outcomes, $\delta_i$ is a pair $(\mathbf{x}', p)$.

- $R$ is the reward model: $R^a(\mathbf{x})$ is the reward for taking action $a$ in state $\mathbf{x}$.

The objective is to maximize the expected total reward of a finite-horizon plan. The DP update for this model is:

$$V^{n+1}(\mathbf{x}) = \max_{a \in A}\{R^a(\mathbf{x}) + \sum_{\mathbf{x}' \in \Delta^a_{\mathbf{x}}} P^a(\mathbf{x}', \mathbf{x}) V^n(\mathbf{x})\} \tag{4.1}$$

where $V^n(\mathbf{x})$ is the value function over the horizon of $n$ time-steps and $V^0(\mathbf{x}) = 0$.

## 4.3 Piece-wise constant model

The structure that we exploit in this chapter consists of partitioning the continuous state space into discrete regions, each of which can be treated as a single entity or an abstract state. In particular, we consider (hyper-)rectangular partitions of the state space $[0,1)^d$. We will use the term "rectangle" or "region" instead of "hyper-rectangle" for brevity, and will discuss examples from a 2-dimensional state space. The formalism generalizes naturally to arbitrary number of dimensions.

The important property of the models is that they are closed under the DP update operator in Equation 4.1. There are many models that satisfy this property; we consider a piece-wise constant model in this section, and a piece-wise linear model in the next section.

**Definition 1 Rectangular Partition** *A rectangular partition of the state space* $[0,1)^d$ *is a finite set of rectangles* $\boxplus = \{\square_1, \square_2, \ldots, \square_k\}$, *where* $\square_i = \prod[X_i^{low}, X_i^{high})$, *such that* $\bigcup_{1 \leq i \leq k} \square_i = [0,1)^d$, *and* $\square_i \cap \square_j = \phi$ *iff* $i \neq j$.

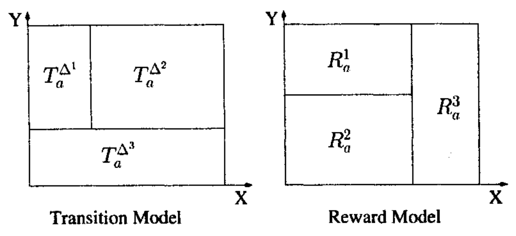


**Figure 4.3.** Rectangular piece-wise constant models

**Definition 2 RPWC function** *A function* $f : [0,1)^d \to \mathbb{O}$ *is RPWC, if there exists a rectangular partition* $\boxplus = \{\square_1, \square_2, \ldots, \square_k\}$ *such that* $\forall i, 1 \le i \le k$, *and* $\forall \mathbf{x}, \mathbf{y} \in \square_i$, $f(\mathbf{x}) = f(\mathbf{y})$.

The set $\mathbb{O}$ that a RPWC function maps to can either be the set of real numbers $\mathbb{R}$, in the case of the reward model, or the set of all possible outcome sets $\Delta$, in the case of the transition model.

As shown in Figure 4.3, the state space is partitioned into rectangular regions. For each action, the outcome set and the probability distribution over it are the same for all states inside a region of the transition model partition. We will use $\Delta^a_\square$ to refer to the outcome set associated with a rectangle $\square$ and an action $a$. In the case of a relative outcome set, for a region $\square$,

$$\forall_{\mathbf{x},\mathbf{y} \in \square} P^a(\mathbf{x} + \delta\mathbf{x}, \mathbf{x}) = P^a(\mathbf{y} + \delta\mathbf{x}, \mathbf{y}).$$

Thus a relative outcome can be seen as *shifting* a region. An absolute outcome maps states in a region to a single state $\mathbf{z}$:

$$\forall_{\mathbf{x},\mathbf{y} \in \square} P^a(\mathbf{z}, \mathbf{x}) = P^a(\mathbf{z}, \mathbf{y}).$$

We will concentrate on the relative transition models, since they are more interesting from a formal and algorithmic standpoint. We will mention implications of absolute models where necessary.

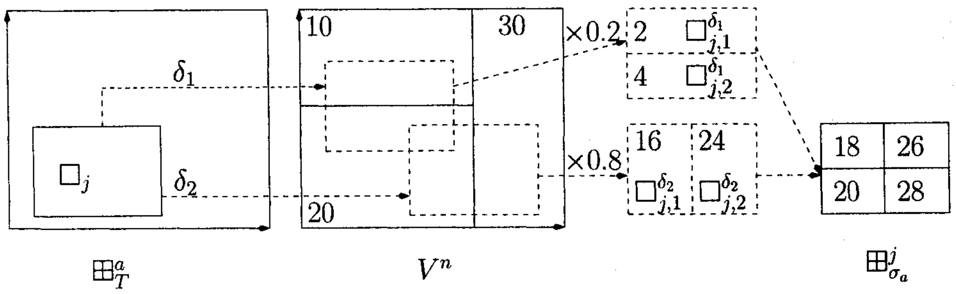


**Figure 4.4.** Computing $\sigma_a$

For a specific action $a$, the transition model is represented by a partition $\boxplus_T^a$, and for each rectangle $\square \in \boxplus_T^a$, a set of relative outcomes $\Delta_\square^a$ together with a probability distribution over these outcomes. Similarly, the rewards $R^a$ are constant in each region. The reward model is represented by a rectangular partition $\boxplus_R^a$ and for each rectangle $\square$ a constant $R_\square$ representing the reward. Note that the partitions for the transition and reward model of an action need not be the same.

Applying RPWC assumptions to the standard model described in the previous section results in an MDP **M1** = $\{\mathbf{X}, A, T_\boxplus, R_\boxplus\}$, where $T_\boxplus$ and $R_\boxplus$ are RPWC transition and reward models as described above. Under this model, if $V^n$ is RPWC, then $V^{n+1}$ computed by the DP update in Equation 4.1 is also RPWC. Since we can represent a RPWC function exactly using a set of rectangles, this theorem enables us to carry out the Bellman backup exactly.

### 4.3.1 Dynamic programming for the piece-wise constant model

We now describe the Bellman backup procedure for the RPWC model. We first show how to compute the summation in Equation 4.1, which we denote as $\sigma_a$:

$$\sigma_a := \sum_{\mathbf{x}' \in \Delta_{\mathbf{x}}^a} P^a(\mathbf{x}', \mathbf{x}) V^n(\mathbf{x}')$$

We construct a partition for $\sigma_a$ by projecting the partition defined by the transition model of action $a$, namely $\boxplus_T^a$, onto the partition defined by $V^n$, using Procedure $\sigma_a$ listed in

Figure 4.1. As an example, Figure 4.4 shows the sub-dividing process for a single rectangle $\square \in \boxplus_T^a$. There are two relative outcomes for this action if taken in $\square$, namely $\delta_1$ with probability 0.2 and $\delta_2$ with probability 0.8. For each outcome, we compute the new position of rectangle $\square$, and intersect it with the partition of $V^n$. The result is then multiplied by the probability of the outcome. Finally, the results of all outcomes are intersected and the summation is computed within each sub-region of the intersection.

Note that this process assumes relative outcomes. For absolute outcomes, we need to modify step 1(a) so that the region $\square_j$ is not subdivided, and is assigned the value of the outcome state in $V^n$ multiplied by the outcome probability.

The remainder of the Bellman backup involves adding the reward and performing the max over all possible actions. The full algorithm is listed as Procedure *Bellman backup* in Figure 4.1. In the whole process, a rectangle is further sub-divided only when necessary during the process of intersecting two partitions.

### 4.3.2 KD-tree representation

For our implementation, we use kd-trees [49] to store and manipulate the rectangular partitions. A kd-tree is a multidimensional generalization of the binary tree in which space is recursively split by hyper-planes orthogonal to one of the $k$ axes. Note that the partition induced by a kd-tree may contain unnecessary splitting of regions in a RPWC function. On the other hand, the intersection operation, which is the main computation of the whole algorithm, can be performed efficiently using algorithms such as [82] on kd-trees. (See Appendix A for a review of the algorithm). Notably, these algorithms treat different numbers of dimensions in a uniform way and have complexities that only depend on the size of the kd-trees.

Actions that have a relative effect on some variable tend to cause the partition of the value function to get finer as the horizon increases, which can affect the efficiency of the algorithm. For this reason, it can be necessary to implement a merging mechanism to unify

```
Procedure $\sigma_a$
  1. For each region $\square_j$ in $\boxplus_T^a$
     (a) For each outcome $\delta_i \in \Delta_{\square_j}^a$
         i. Compute the region $\square_j^{\delta_i}$ resulting from shifting $\square_j$ by the relative outcome $\delta_i$.
         ii. Intersect the shifted region $\square_j^{\delta_i}$ with the partition of $V^n$, producing sub-regions $\square_{j,k}^{\delta_i}$
         iii. Assign to each sub-region $\square_{j,k}^{\delta_i}$ the value of the corresponding region of $V^n$ multiplied by the probability of the outcome $\delta_i$.
     (b) Intersect all the shifted regions from all of the outcomes, producing partition $\boxplus_{\sigma_a}^j$.
     (c) Assign to each of the regions in partition $\boxplus_{\sigma_a}^j$ the sum of the values of the corresponding sub-regions $\square_{j,k}^{\delta_i}$.
  2. Assemble the final partition: $\boxplus_{\sigma_a} = \cup_j \boxplus_{\sigma_a}^j$.
Procedure Bellman backup
  1. Compute partition $\boxplus_{\sigma_a}$ for all $a$ using Procedure $\sigma_a$.
  2. For each action $a$
     (a) Intersect partition $\boxplus_{\sigma_a}$ with $\boxplus_R^a$ to get partition $\boxplus_{Q_a}$.
     (b) The value of each region in $\boxplus_{Q_a}$ is computed by summing the values of the corresponding regions of $\boxplus_{\sigma_a}$ and $\boxplus_R^a$.
  3. The partitions $\boxplus_{Q_a}$ of all actions are intersected, producing $\boxplus_{V^{n+1}}$.
  4. The value of each region in $\boxplus_{V^{n+1}}$ is computed as the max of each of the corresponding regions in all the partitions $\boxplus_{Q_a}$.
```

**Table 4.1.** Dynamic programming for the piece-wise constant model

neighboring regions with the same value. One can further reduce the complexity of the algorithm by merging pieces with similar value, trading quality of solution for computation time.

Merging based solely on value can break the RPWC property, resulting in partitions that can not be represented by kd-trees. For our implementation, we performed the merging taking into account both the value and the structure of the kd-tree representation, by performing a depth-first traversal of the kd-tree, and merging the leaf-nodes of the kd-tree if they have the same value. This way the kd-tree representation is maintained throughout the merging process.

## 4.4 Piece-wise linear and convex model

In this section, we extend the model **M1** by allowing more complex reward structures so that richer domains can be modeled. For example, to take into account the lighting of a rock from the sun in a rover problem, the value of taking a picture could vary linearly with time of the day. To model such structures, we extend the RPWC reward model to a *rectangular piece-wise linear and convex* (RPWLC) reward model.

**Definition 3 RPWLC** *A function* $f : [0,1)^d \rightarrow \mathbb{R}$ *is RPWLC if 1) there exists a rectangle partition* $\boxplus = \{\square_1, \ldots, \square_k\}$*; and 2)* $\forall i, 1 \leq i \leq k$*, there exists a PWLC function* $L_i$ *such that* $\forall \mathrm{x} \in \square_i, f(\mathrm{x}) = L_i(\mathrm{x})$.

This representation allows $R$ and $V^i$ for a region of the partition to be the maximum of a set of linear functions, rather than a single function as in **M1**. We allow this because a Bellman backup will create non-rectilinear regions when it performs the maximization step on linear functions.

We will refer to the model with the RPWLC assumption as **M2** $= \{\mathbf{X}, A, T_{\boxplus}, R^L_{\boxplus}\}$, where $R^L_{\boxplus}$ is the RPWLC reward model as defined above. Often, the reward function may be of a simpler form (a single linear function per region), but the resulting value function will remain RPWLC. Note that the transition model remains RPWC in M2.

As in the RPWC model **M1**, the operations during the Bellman backup for **M2** are intersection, summation, and max. The intersection is identical. The difference between **M1** and **M2** arises from the fact that for the value functions in **M2**, each rectangle contains a set of linear functions rather than a single scalar value. In particular, we need to perform *addition* and *maximization* between two sets of linear functions over the same rectangle. These operations are well defined in standard POMDP algorithms as reviewed in Section 2.6. In particular, the addition of two sets of linear functions $L_1$ and $L_2$ is carried out by the *cross-sum* operator, and the maximization be carried out by the *union* operator.

Just as in standard POMDP algorithms, pruning is performed to remove dominated linear functions. Maximization over the remaining linear functions is used to determine the value and policy of any point in the state space. This considerably improves the efficiency of the algorithm, and the combination of pruning and maximization over the remaining linear functions achieves the maximization part of the Bellman equation.

To adapt the algorithms in Table 4.1 so that they support the **M2** model, we make the following changes:

- In step (1)(c) of Procedure $\sigma_a$ and step (2)(b) of Procedure *Bellman backup*, we change sum of values to cross-sum of linear functions.
- In step (4) of Procedure *Bellman backup*, we change max of values to union of linear functions.
- Then we add a pruning after step (1)(c) of Procedure $\sigma_a$ and after steps (2)(b) and (4) of Procedure *Bellman backup*.

## 4.5 Mixed model

In this section, we further extend the model to include discrete state components, which is the case of the Mars rover domain that motivated this work [23]. The new model is defined as **M3**$= \{S, \mathbf{X}, A, T_{\boxplus}, R^{L}_{\boxplus}\}$, where $S$ is a set of discrete states. The full state space is the product $S \times \mathbf{X}$. We will use $(s, \mathbf{x})$ to refer to a specific state in the state space. $P^a(s', \mathbf{x}', s, \mathbf{x})$ is the probability of reaching $(s', \mathbf{x}')$ if action $a$ is taken in state $(s, \mathbf{x})$. We will generally define the transition model by a a marginal probability distribution on the arrival discrete state: $P^a_d(s', s, \mathbf{x})$, and a conditional distribution over the continuous space $P^a_c(\mathbf{x}', s', s, \mathbf{x})$ given the arrival discrete state. As in **M1** and **M2**, the conditional distribution $P_c^a$ is assumed to be RPWC. For reward, we will define a function $R_s^a(\mathbf{x})$ for each action and discrete state pair, and assume that all $R_s^a$ are RPWLC. We represent the value function over the full state space using a set of functions $V := \{V_s(\mathbf{x}) | s \in S\}$. The Bellman backup for **M3** can be performed as:

$$V_s^{n+1}(\mathbf{x}) = \max_{a \in A}\{R_s^a(\mathbf{x}) + \sum_{s'} P_d^a(s'|s, \mathbf{x})\sigma_a^{s'}\}$$

$$\sigma_a^{s'} = \sum_{\mathbf{x}'} P_c^a(\mathbf{x}', s', s, \mathbf{x}) V_{s'}^n(\mathbf{x}')$$

Algorithmically, the addition of discrete states changes the backup procedure by adding a loop over the discrete states to the computation described in the previous section.

## 4.6 Empirical evaluations

This section presents empirical results obtained by solving prototype instances of the Mars rover domain. This line of research is being continued at NASA at the time of writing of this thesis, with new results being reported recently [78, 77].

We tested our algorithms on a Mars rover domain adapted from [23]. The domain contains a "primary" plan, which consists of approaching a target point, digging the soil, backing up, and taking spectral images of the area. All these actions are assumed to consume time and battery power according to different Gaussian distributions. There are two potential branches to the primary plan: The first branch is to replace the spectral imaging with a high-resolution camera imaging, which in general consumes more time and energy. The second branch is to replace the digging-backing-imaging plan with a simple low-resolution imaging, and then perform on-board image analysis.

We model the domain with 11 discrete states representing different stages of the rover exploration process. We vary the number of continuous variables, which model different types of resources, from 1 to 3, creating three sets of test problems, referred to as 1D, 2D and 3D, respectively. In the original domain, action effects on the resources are modeled

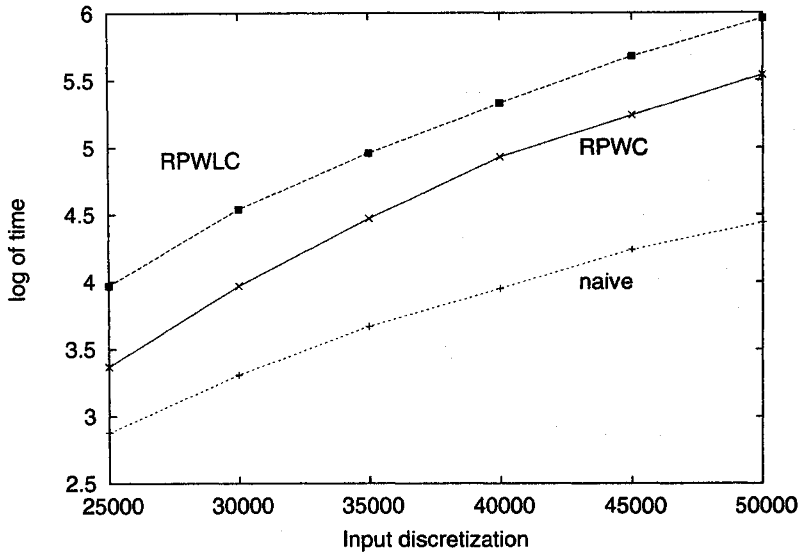


**Figure 4.5.** Result on problem set 1D.

with continuous probability distributions. In our model, these continuous distributions are discretized. The resolution of the discretization is the independent variable in our experiments. For each resolution, we create two versions of the problem, one with constant rewards (RPWC representation), the other with rewards that are linear functions of the continuous variables (RPWLC representation). We compare the performance of our algorithm on each of these two representations with a naive algorithm that discretizes the value function using the same resolution used in the input discretization.

Figures 4.5 to 4.7 show the results. The $X$-axis shows the input discretization resolution on each continuous variable. The $Y$-axis shows the elapsed run-time of the different algorithms, on a logarithmic scale. Note that for the naive approach, the run-time of the two versions (RPWC and RPWLC) of the problem are largely the same, because after the discretization of the value function, the identical amount of computation is carried out for both problems. Thus only the result on the RPWC problem is plotted.

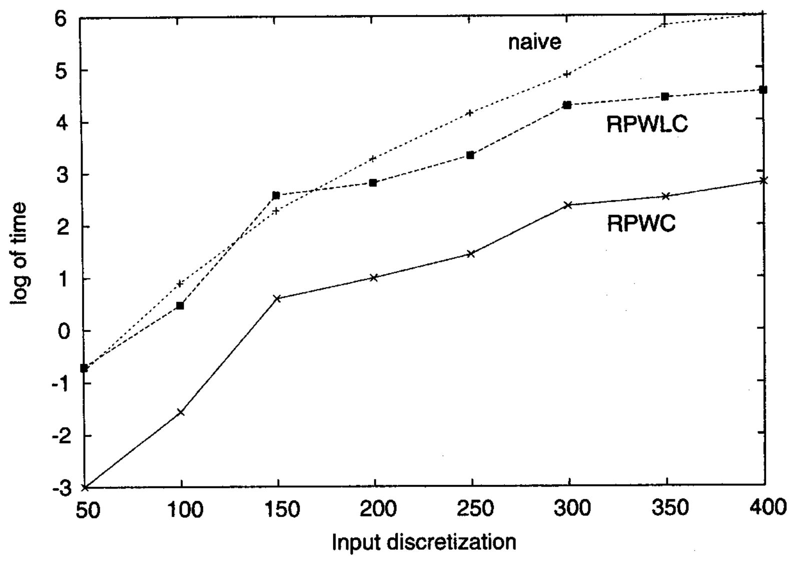


**Figure 4.6.** Result on problem set 2D.

As the figures show, our algorithm is slower than the naive approach for all the 1D problems. The overhead of dealing with the complex data structure exceeds the savings gained from it for the simple version of the problems. For the 2D problems, the RPWC model outperforms the naive approach. For lower input resolutions (from 50 to 150), the RPWLC model performs similarly to the naive approach. However, it is considerably faster for higher resolution problems. For the 3D problems, the difference between the naive approach and our approaches is more dramatic. In particular, the naive approach did not finish after 3 hours for problems with resolution greater than 80.

These results shows that our algorithm can scale better as the number of continuous variables increases. The improved scalability results from exploiting the specific problem structure to avoid unnecessary discretization of the value function. Figure 4.8 shows the resulting value function on a specific discrete state of a problem in the 2D set with RPWLC rewards, and an input discretization of 25 on each dimension. The left side shows the actual

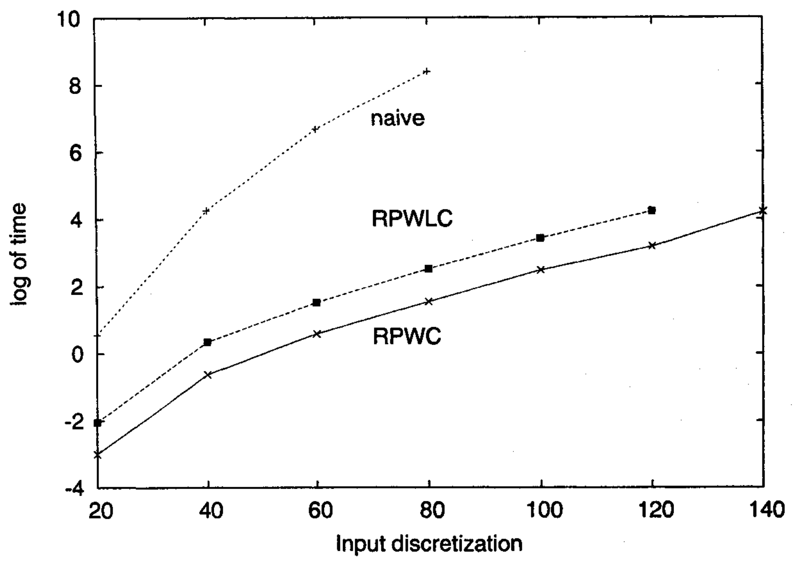


**Figure 4.7.** Result on problem set 3D.

function, and the right side shows the corresponding partition over the continuous space. As we can see, fine discretization is only applied to the upper right region. Approximately 70% of the space is treated exactly with only a small number of regions. In contrast, the naive approach discretizes the entire space evenly, expending a large amount of computation on areas that are, in fact, from the same linear function. Our algorithm avoids this computation by treating large regions as a single state.

Two features in Figure 4.8 deserve further analysis. Firstly, although the input is only discretized with a resolution of 25, the resulting partition has considerably more discretization points, albeit all concentrated at the upper right region. This is because the initial partitions defining the transition and reward models are not necessarily aligned with the input discretization, so a finer partition is needed to represent the optimal value function. The naive approach doesn't make the additional distinctions so it misses details of the value

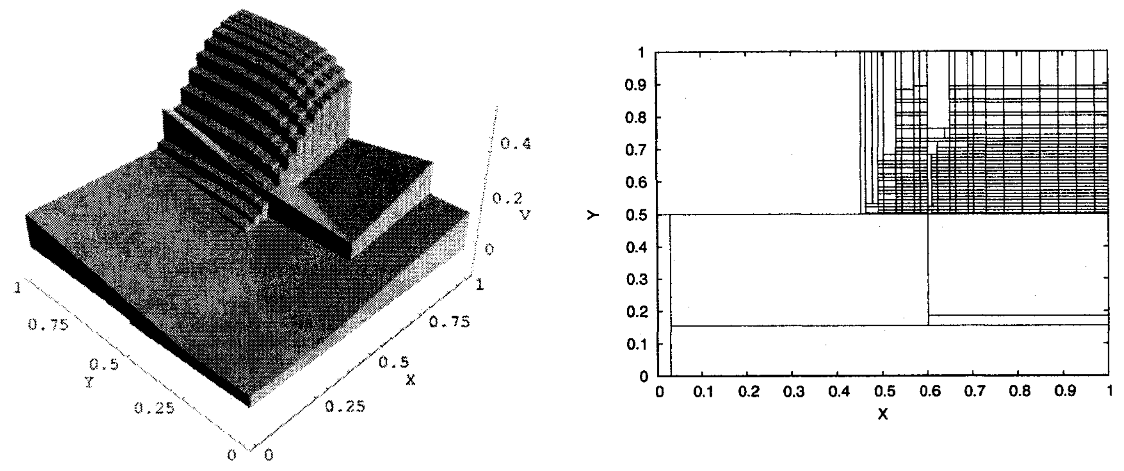


**Figure 4.8.** The piece-wise linear value function and the corresponding space partition for the starting discrete state in a 2D problem.

function. The complexity grows over the course of the dynamic programming, so early iterations use coarser partitions, providing another saving over the naive approach.

Secondly, note the region from around point $(0.6, 0.2)$ to $(1.0, 0.5)$. The value function as can be seen on the left of Figure 4.8 over this region has a curved shape. It is in fact composed of 13 linear functions. This is typical when the reward model is RPWLC. Again, the dynamic programming is keeping the discretization to a minimum by automatically grouping states whose value function can be represented in a single PWLC form into an abstract state.

For all tests, the solution times for RPWLC models are greater than those of the RPWC model. This is because of the extra computation on the linear vectors in the RPWLC model, in particular, solving the linear programs to keep the representation of the PWLC function minimal. For the 3D problem set, the RPWLC model runs out of memory for the problem with input discretization of 140. The primary cause of this is that some regions require a large number of linear vectors to represent the value function. Our current algorithm attempts to minimize the number of regions. However, we can introduce a trade-off between the size of the partition and the size of the vector representation, by sub-dividing a partition to allow more vectors to be pruned. The idea that smaller partition allows more vectors to be pruned will be fully explored in Chapter 6, when we deal with the vector pruning issue in solving POMDPs.

## 4.7 Chapter notes

### 4.7.1 Contributions

The technical contribution in this chapter is a novel state abstraction algorithm for performing dynamic programming in solving a class of continuous state MDPs. Encouraged by the results, NASA engineers are extending the models and techniques developed in this chapter to more challenging problems [78, 77].

Within this thesis, this chapter serves an important role. It demonstrates one way of explicitly representing regions of a continuous state space. As pointed out in Chapter 2, representation of the continuous belief space is one of the major challenge in POMDP algorithms. Although the representation developed later in Chapter 6 for POMDPs is vastly different than the rectangle representation used in this chapter, they share conceptual similarities in that they partition the continuous space into a finite number of regions. Further, the PWLC representation us used in this chapter is a direct adaption from POMDP algorithms.

### 4.7.2 Related work in exploiting state abstraction

Although the model considered in this chapter involves continuous state variables, the state abstraction process is very similar to the symbolic approach used in Chapter 3 and [37, 59]. In the symbolic approach, value functions are represented by decision diagrams (or decision trees in the case of [37]) that maps discrete state variables to values. Here value function over a continuous state space is represented by a KD-tree that partitions the state space into rectangle regions, where each region can be treated as an abstract state. Dynamic programming is then carried out on top of this representation, computing a new KD-tree representing the updated value function given another KD-tree representing the current value function.

The choice of a rectangle partitioning scheme may seem overly restrictive. However it makes the data structure easy to handle, with efficient algorithms readily available from the computational geometry community. Further, the ability to define piece-wise linear and convex functions over a region substantially increases its modeling power.

To the best of my knowledge, this is the first general approach toward state abstraction in continuous MDPs. The closest related previous work along this line is probably the Explanation-Based Reinforcement Learning (EBRL) framework by Dietterich and Flann [38]. In it, a "region-based" dynamic learning algorithm is developed that uses rectangles to represents regions of a discrete grid world that have identical values. It differs from this work in the following aspect: 1) EBRL is designed for discrete grid-world state spaces, not continuous state spaces; 2) EBRL is a goal regression algorithm. Given a rectangle goal region, it reasons about what other rectangle regions can be constructed that will reach the goal region, hence the "explanation" component of the algorithm. 3) EBRL only allows constant value over a rectangle region. Another closely related work is the time-dependent MDP model by Boyan and Littman [20]. It can be seen as a special case of the model considered in this chapter, where only one continuous state variable is allowed. The regions in this case are one-dimension intervals, which are easier to handle than the multi-dimensional regions considered in this chapter.

### 4.7.3 Related work in continuous state MDPs

The most common approach to handle continuous state variables in MDPs is to use function approximators such as artificial neural networks[10, 98], to discretize the continuous state space more or less naively, which does not scale well to multiple dimensions, or to use Monte-Carlo approaches [99]. None of these approaches exploits the structure in the problem. In fact, the initial attempt to solve the problem as reported in [23] uses a variant of the Monte-Carlo algorithm to compute the function in Figure 4.2; the algorithm took orders of magnitude longer than the method in this chapter. Munos and Moore [81] propose a formal model of a continuous MDP and algorithms for discretizing it adaptively. Their approach involves solving the MDP at one level of discretization, then locally refining the discretization, and repeating until the approximation is good enough. In contrast, our algorithm finds the correct level of discretization and solves the MDP only once, while at the same time takes advantage of the structure of the problem.

# CHAPTER 5

# DISCRETE STATE ABSTRACTION FOR POMDPS

From this chapter on, we turn our attention to the more general POMDP model. The main difficulty in solving POMDPs is the PWLC representation of the value function over the belief space, which is a continuous simplex with a dimensionality equal to the size of the original discrete state space of the POMDP. This chapter describes a simple state abstraction technique for the discrete state space so that the dimensionality of the belief space can be reduced. The approach is a natural generalization of the symbolic abstraction techniques presented in Chapter 2. However, the benefit of the abstraction is somewhat different than in the fully observable model. The results in this chapter shows the effectiveness of abstraction on the discrete state space for POMDPs, but also exposes its limitation and motivates the research to study abstraction directly in the continuous belief space in the next chapter.

## 5.1 Symbolic abstraction for POMDPs

Like the discrete state space of the MDP model described in Section 2.5.2, the discrete state space of a POMDP can also be represented symbolically using a set of Boolean variables. In this chapter, we consider the following symbolic POMDP model which is an extension of the symbolic MDP model reviewed in Section 2.5.2:

- $\mathbf{X} = \{X_1, \ldots, X_k\}$ is a finite set of Boolean state variables;
- $Z$ is a finite set of observation states;

- $P$ is the transition model. $P^a(\mathbf{X}'|\mathbf{X})$ is an ADD that represents the transition function of action $a$;
- $O$ is the observation model. $O^a(z|\mathbf{X}')$ is an ADD that represents the probability of observing $z$ if action $a$ is taken and resulted in a transition to a state described by $\mathbf{X}'$.
- $R$ is the reward model. $R^a(\mathbf{X})$ is an ADD representing the reward function of action $a$.

Recall that the value function of a POMDP is a piece-wise linear and convex (PWLC) function defined over the *belief space*, a probability distribution over the discrete state space of the POMDP (Section 2.6). A PWLC function $V$ is represented by a set of linear vectors $\mathcal{V} = \{v_1, \ldots, v_m\}$. Each vector is a mapping from the state space to some values. Therefore each vectors can be represented by an ADD, denoted $v_i(\mathbf{X})$. The PWLC function is in turn represented by a set of ADDs.

## 5.2 Symbolic dynamic programming

As reviewed in Section 2.6, the dynamic programming update for POMDP involves constructing a new set of vectors $\mathcal{V}^{n+1}$ that represents the $(n+1)$-step-to-go value function, from an existing set of vectors $\mathcal{V}^n$ that represents the $n$-step-to-go value function. The DP update process begins by constructing a *projection* set for each pair of action and observation:

$$\{v^{a,z,i} | v^i \in \mathcal{V}^n\},$$

where

$$\forall s, v^{a,z,i}(s) = \frac{R^a(s)}{|Z|} + \beta \sum_{s' \in S} O^a(z|s')P^a(s'|s)v^i(s')$$

is the projection computation. As in the symbolic dynamic programming for fully observable MDPs, this computation can be carried out using ADD computation as follow:

$$v^{a,z,i}(\mathbf{X}) = \frac{R^a(\mathbf{X})}{|Z|} + \beta \sum_{\mathbf{X}'} O^a(z|\mathbf{X}')P^a(\mathbf{X}'|\mathbf{X})v^i(\mathbf{X}').$$

Here, the vectors $v^{a,z,i}$ and $v^i$, the transition function $P^a$, the observation function $O^a$, and the reward function $R^a$ are all represented by ADDs. All binary operations such as addition, multiplication, and division are carried using ADDs. The summation over $\mathbf{X}'$ is the existential abstraction operator a reviewed in Section 2.5.2. The benefit of using ADDs to perform this computation is the same as in the MDP case: The ADD exploits state abstractions by grouping states with the same value together and treat them as a single abstract state. When there are large number of states in a vector that has the same value, the ADD computation can be much more efficient than a flat state computation.

## 5.3 Pruning in abstract state space

Although the ADD representation helps speedup the projection computation, it's main benefit lines in the pruning computation. Recall that after the projection, three pruning steps are carried out to construct the $(n+1)$-step value function:

$$\begin{aligned}
\mathcal{V}^{a,z} &= \mathbb{PR}\{v^{a,z,i} | v^i \in \mathcal{V}\}; \\
\mathcal{V}^{a} &= \mathbb{PR}\left\{\oplus_{z \in Z} \mathcal{V}^{a,z}\right\}; \\
\mathcal{V}' &= \mathbb{PR}\left\{\cup_{a \in A} \mathcal{V}^{a}\right\}.
\end{aligned}$$

Each pruning takes as input a set of vectors $\mathcal{U}$ and remove those in $\mathcal{U}$ that are dominated. Most dominated vectors can only be detected by solving the linear program listed in Table 2.1 (page 22).

To use this linear program to prune a set of vectors represented in factored form, we first perform the pre-processing step summarized in Table 5.1. It takes as input a set of ADDs $\mathcal{V}$ and creates an abstract state space for it that only makes the state distinctions according to the value they map to. Each ADD defines an abstraction of the state space where the number of abstract states is equal to the number of leaves of the ADD, and each abstract state represents a set of states with the same value. The set of abstract states defined by and

```
procedure CREATE-PARTITION($\mathcal{V}$)
$R \leftarrow \{S\}$
for each vector $v \in \mathcal{V}$
    $T \leftarrow$ set of abstract states defined by $v$
    for each abstract state $t \in T$
        for each abstract state $r \in R$
            if ($t \cap r = \emptyset$) or ($r \subseteq t$) then do nothing
            else if $t \subset r$ then
                $R \leftarrow R - \{r\}$
                $R \leftarrow R \cup \{t\} \cup \{r - t\}$
                exit innermost for loop
            else
                $R \leftarrow R - \{r\}$
                $R \leftarrow R \cup \{r - t\} \cup \{r \cap t\}$
                $T \leftarrow T \cup \{t - r\}$
                exit innermost for loop
return $R$
```

**Table 5.1.** Algorithm for partitioning a state set, $S$, into a set of abstract states, $R$, that only makes relevant state distinctions found in a set of ADDs $\mathcal{V}$.

ADD can be constructed by first enumerating the leaves of the ADD. For each leaf $L$, a new ADD that represents the characteristic function corresponding $L$ is constructed by first copying the original ADD and then setting the leaf $L$ to 1 and all other leaves to 0. This is the same procedure used to construct the generalization set used in the symbolic RTDP algorithm described in Section 3.3.1.

In the pseudocode for CREATE-PARTITION, $R$ denotes a set of abstract states. Initially, $R$ contains a single abstract state that corresponds to the entire state set of the problem. Gradually, $R$ is refined by making relevant state distinctions. Set operations such as union and difference are implemented using set operations in ADDs (or BDDs). Each new state distinction splits some abstract state into two abstract states. The algorithm does not backtrack; every state distinction it introduces is necessary and relevant. The algorithm has a worst-case running time of $|\mathcal{V}||S|^2$, although it runs much faster when there is state abstraction. (It is easy to imagine a faster heuristic algorithm that finds a useful, though not necessarily best, state abstraction. For the examples we have tested so far, we have not found this necessary.)

In the resulting abstraction, two states will be mapped to the same abstract state if they have the same value in each of the ADDs in $\mathcal{V}$. Because an abstract state corresponds to a set of underlying states, the cardinality of the abstract state space can be much less than the cardinality of the original state space. Using this abstract state space, the number of variables in the linear program can be greatly reduced, leading to reduced computational time solving the linear programs. As we will see in the next section, solving these linear programs takes most of the computation time in the DP update process for POMDPs.

## 5.4 Empirical evaluation

We test the effectiveness of the state abstraction technique using the incremental pruning (IP) algorithm [25]. One version of IP uses the symbolic representation and solves the pruning linear programs in abstract state space. The other version is the original algorithm that does not use a symbolic representation and solves the linear programs in the original flat state space.

Table 5.2 shows some representative timing results. For six different POMDPs, it compares one iteration of incremental pruning with and without the state abstraction technique described in previous sections. The results without state abstraction are shown above the results with state abstraction. It is important to note that both algorithms have the same input and compute identical results. Both perform the same number of projections, the same number of pruning operations, the same number of linear programs, etc. The main difference is the data structures they use for the projection and the state space used to construct the linear programs used for pruning.

The column with the heading "Abs" indicates the average cardinality of the abstract state sets that are created. In general, the higher the degree of abstraction for a problem, the faster the symbolic algorithm runs relative to the original algorithm. The relationship

|  | Problem characteristics |  |  | Solution characteristics |  |  | Timing Results |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| # | $\|S\|$ | $\|A\|$ | $\|O\|$ | $\|\mathcal{V}\|$ | $\|\mathcal{V}'\|$ | Abs | projection | partition | LP | total |
|  |  |  |  |  |  | - | 0.37 | - | 480.64 | 498.70 |
| 1 | $2^5$ | 3 | $2^1$ | 91 | 229 | 14.0 | 1.06 | 9.76 | 343.69 | 366.31 |
|  |  |  |  |  |  | - | 0.21 | - | 3640.28 | 3831.83 |
| 2 | $2^5$ | 6 | $2^2$ | 25 | 142 | 20.9 | 0.21 | 8.63 | 3471.36 | 3615.33 |
|  |  |  |  |  |  | - | 1.14 | - | 5.37 | 6.69 |
| 3 | $2^6$ | 6 | $2^2$ | 32 | 30 | 7.5 | 0.63 | 0.60 | 2.69 | 4.04 |
|  |  |  |  |  |  | - | 53.42 | - | 7957.78 | 8181.89 |
| 4 | $2^6$ | 5 | $2^2$ | 521 | 2539 | 64.0 | 395.63 | 1.81 | 8072.14 | 8653.18 |
|  |  |  |  |  |  | - | 7.53 | - | 132.96 | 154.18 |
| 5 | $2^7$ | 8 | $2^2$ | 42 | 42 | 12.8 | 5.57 | 3.81 | 30.49 | 41.44 |
|  |  |  |  |  |  | - | 4113.07 | - | 8099.05 | 12546.14 |
| 6 | $2^{10}$ | 11 | $2^3$ | 198 | 457 | 34.6 | 225.31 | 63.36 | 641.75 | 959.67 |

**Table 5.2.** Representative timing results (in CPU seconds) for an iteration of the dynamic programming update using incremental pruning with and without state abstraction.

between degree of abstraction and computational speedup is not quite so simple, however. It can be complicated by other factors, including the cardinality of the set of state-value functions and the number of observations. We have broken down the timing results to help illustrate the effect of these various factors. (Note that the total time is slightly larger than the sum of the times for all the operations; this reflects a slight overhead in the algorithm for setting up the computation.)

Problem 1 in Table 5.2 is the coffee problem used as an illustration in [17]. Problem 3 is a variation of the widget-processing example in [40]. The other problems are various synthetic POMDPs created for the purpose of testing. Problem 4 illustrates the worst-case behavior of the algorithm when there is no state abstraction. Problem 2 illustrates the performance of the algorithm where there is a very modest degree of state abstraction. Problems 5 and 6 illustrate the performance of the algorithm when there is a high degree of state abstraction; they represent close to the best-case behavior of the algorithm for problems of their size. As problem size increases, that is, as the cardinality of $\mathcal{S}$, $\mathcal{O}$, and $\mathcal{V}$ increases, the potential speedup from using a factored representation increases as well.

Two operations of the dynamic programming algorithm exploit a symbolic representation: the projection operation and the linear programming (LP) in the pruning operation. The table breaks down the timing results for operation. For the symbolic algorithm, the time used for creating the abstraction is also listed in the column "partition".

For most POMDPs, and for all POMDPs that are difficult to solve, most of the running time of dynamic programming is spent pruning vector sets. Our results show that solving linear programs constructed over the abstract state space is significantly faster than solving linear programs constructed over the original state space. This is the major contribution of the abstraction technique to the whole algorithm. Further, the only overhead for the algorithm is the creating an abstract state space. Empirically, we found that this overhead is quite small relative to the running time of the rest of the pruning algorithm. It tends to grow with the size of the abstract state space, of course. Nevertheless, problem 4, our worst-case example, does not illustrate the worst-case overhead. This is because the procedure CREATE-PARTITION has a useful and reasonable optimization; as soon as the cardinality of the abstract state space is equal to the cardinality of the underlying state space, it terminates. For CREATE-PARTITION to incur its worst-case overhead, in other words, there must be at least some state abstraction. But this, in turn, can offset the overhead.

The projection operation for POMDPs is very similar to the symbolic dynamic programming update for fully observable MDPs reviewed in Section 2.5.2. For this operation, our results is consistent with those reported in [59]. We found that its worst-case overhead can cause a symbolic projection to run up to ten times slower than a regular projection. This is illustrated by problem 4. But also like [59], we found that symbolic projection can run much faster than regular projection when there is sufficient state abstraction. This is illustrated by problem 6, in particular.

## 5.5 Limitation of discrete state abstraction for POMDPs

The positive result reported in this chapter must be placed in perspective. The approach described addresses POMDP difficulty that is due to the size of the state space. For completely observable MDPs, the size of the state space is the principal source of problem difficulty. For POMDPs, it is not. A POMDP with a large state space may have a small optimal value function that can be found quickly. By contrast, a POMDP with a small state space may have a large or unbounded value function that is computationally prohibitive to compute. It is the size of the value function representing a solution, not the size of the state space, that primarily reflects POMDP difficulty. Although the framework described in this chapter for exploiting a symbolic state representation can be very beneficial, it does not address this more important aspect of POMDP difficulty, which is the main topic of the next chapter.

The fact that the symbolic representation mainly benefits the pruning operator but not the projection operator suggests a simpler approach toward state abstraction for POMDPs without using a symbolic representation (and therefore forgo the limited benefit it has on the projection step). In [45], we report results on a simpler state abstraction algorithm that does not rely on a symbolic representation. Instead, it uses a flat state representation, and builds the partition by walking through the states one by one. As the results in [45] shows, this resulted in more computational time on the projection operation and on creating the abstract state space. However, it gives the same reduction to the pruning step. As a whole the simpler abstraction algorithm still presents a significant speedup over algorithms that does not exploit state abstraction at all.

The discrete state space and the size of value function representation is not completely independent, either. By ignoring smaller differences among the state values in each vector, more vectors can be pruned. This is essentially an approximate way of pruning by treating non-dominated vectors as dominated if they only contribute small value differences. The possibility of such approximation is suggested in [28, 17]. In [43, 45], we show theoretical and empirical evidence supporting this using both symbolic and non-symbolic approaches toward state abstraction.

# CHAPTER 6

# REGION-BASED BELIEF STATE ABSTRACTION FOR POMDPS

Because of the difficulty in explicitly representing the uncountably infinite belief state space, previous POMDP algorithms have relied on an implicit representation of the state space which makes exploiting state abstraction difficult. This chapter introduces an explicit, region-based representation of the belief state space. This representation allows us to analyze the POMDP model and algorithms from a unique perspective, and enables us to design new algorithms that are significantly faster than the previous best algorithms. In particular, the cross-sum operation, which is the bottle-neck of the DP update, can be improved exponentially [47]. The whole DP algorithm when integrated with the region-based representation enjoys orders of magnitudes of speedup over previous algorithms [48].

## 6.1 Pruning revisited

Recall that the DP update for POMDPs can be computed in three steps using the vector representation (Section 2.6.2):

$$\begin{aligned}\mathcal{V}^{a,z} &= \mathbb{PR}\{v^{a,z,i} | v^i \in \mathcal{V}\}; &(6.1)\\ \mathcal{V}^{a} &= \mathbb{PR}\{\oplus_{z\in Z}\mathcal{V}^{a,z}\}; &(6.2)\\ \mathcal{V}' &= \mathbb{PR}\{\cup_{a\in A}\mathcal{V}^{a}\}, &(6.3)\end{aligned}$$

where the individual vectors $v^{a,z,i}$ in Equation 6.1 is computed by the projection

$$v^{a,z,i}(s) = \frac{R^a(s)}{|Z|} + \beta \sum_{s'\in S} O^a(z|s')P^a(s'|s)v^i(s'). \qquad (6.4)$$

The $\mathbb{PR}$ operator, applied in all three steps, is the pruning operator that reduces a set of vectors to its minimal form by removing dominated vectors. Equation 6.1 is called *projection pruning*; Equation 6.2 is called *cross-sum pruning*; and Equation 6.3 is called *maximization pruning*.

There are two tests for dominated vectors. The simpler method is to remove any vector $u$ that is point-wise dominated by another vector $w$. That is, $u(s) \leq w(s)$ for all $s \in S$. The procedure POINTWISE-DOMINATE in Table 6.1 performs this operation. Although this method of detecting dominated vectors is fast, it cannot detect all dominated vectors.

There is a linear programming method that can detect all dominated vectors. Given a vector $w$ and a set of vectors $\mathcal{U}$ that does not include $w$, the linear program in procedure LP-DOMINATE of Table 6.1 determines whether adding $w$ to $\mathcal{U}$ improves the value function represented by $\mathcal{U}$ for any belief state $b$. If it does, the variable $d$ optimized by the linear program is the maximum amount by which the value function is improved, and $b$ is the belief state that optimizes $d$. If it does not, that is, if $d \leq 0$, then $w$ is dominated by $\mathcal{U}$.

The algorithm summarized in Table 6.1 uses these two tests for dominated vectors to prune a set of vectors to its minimum size. The symbol $<_{lex}$ in the pseudo-code denotes lexicographic ordering [70].

Since the linear programming method takes up most of the computation time in the pruning, to simplify our discussion, we will omit analyzing the point-wise domination test in the rest of the paper. We can assume either that the point-wise domination test is always performed before pruning since it takes little computation time, or that we don't use the point-wise domination test at all since it can only detect a small number of dominated vectors.

As we can see from the table, to prune a vector set $\mathcal{W}$, we need to solve a linear program for each vector in $\mathcal{W}$. In other words, to prune the set $\mathcal{W}$ we need to solve $|\mathcal{W}|$ LPs. Here the cross-sum pruning (6.1) presents a major bottleneck because the size of the cross-sum is the product of the inputs:

```
procedure POINTWISE-DOMINATE($w, \mathcal{D}$)
1.  for each $u \in \mathcal{D}$
2.      if $w(s) \leq u(s),\ \forall s \in S$ then return true
3.  return false
procedure LP-DOMINATE($w, \mathcal{D}$)
4.  solve the following linear program
        variables: $d,\ b(s)\ \forall s \in S$
        maximize $d$
        subject to the constraints
            $b \cdot (w - u) \geq d,\ \forall u \in \mathcal{D}$
            $\sum_{s \in S} b(s) = 1$
5.  if $d \geq 0$ then return $b$
6.  else return nil
procedure BEST($b, \mathcal{W}$)
7.  $max \leftarrow -\infty$
8.  for each $u \in \mathcal{W}$
9.      if $(b \cdot u > max)$ or $((b \cdot u = max)$ and $(u <_{lex} w))$
10.         $w \leftarrow u$
11.         $max \leftarrow b \cdot u$
12. return $w$
procedure $\mathbb{PR}(\mathcal{W})$
13. $\mathcal{D} \leftarrow \emptyset$
14. while $\mathcal{W} \neq \emptyset$
15.     $w \leftarrow$ any element in $\mathcal{W}$
16.     if POINTWISE-DOMINATE($w, \mathcal{D}$) = true
17.         $\mathcal{W} \leftarrow \mathcal{W} - \{w\}$
18.     else
19.         $b \leftarrow$ LP-DOMINATE($w, \mathcal{D}$)
20.         if $b = nil$ then
21.             $\mathcal{W} \leftarrow \mathcal{W} - \{w\}$
22.         else
23.             $w \leftarrow$ BEST($b, \mathcal{W}$)
24.             $\mathcal{D} \leftarrow \mathcal{D} \cup \{w\}$
25.             $\mathcal{W} \leftarrow \mathcal{W} - \{w\}$
26. return $\mathcal{D}$
```

**Table 6.1.** Algorithm for pruning a set of vectors $\mathcal{W}$.

$$|\mathcal{U} \oplus \mathcal{W}| = |\mathcal{U}| \times |\mathcal{W}|.$$

As a result, it is necessary to process $\prod_z |\mathcal{V}^{a,z}|$ vectors in computing $\mathcal{V}^a$. This translates into solving $\prod_z |\mathcal{V}^{a,z}|$ LPs. Incremental pruning (IP) [25] is designed to specifically addresses this problem. It exploits the fact that the $\mathbb{PR}$ and $\oplus$ operators can be interleaved:

$$\mathbb{PR}(\mathcal{U} \oplus \mathcal{V} \oplus \mathcal{W}) = \mathbb{PR}(\mathcal{U} \oplus \mathbb{PR}(\mathcal{V} \oplus \mathcal{W})). \tag{6.5}$$

Thus Equation (6.1) can be computed as follows:

$$\mathcal{V}^a = \mathbb{PR}(\mathcal{V}^{a,z_1} \oplus \mathbb{PR}(\mathcal{V}^{a,z_2} \oplus \cdots \mathbb{PR}(\mathcal{V}^{a,z_{k-1}} \oplus \mathcal{V}^{a,z_k}) \cdots)), \tag{6.6}$$

which is what the IP algorithm does. The benefit of IP is the reduction of the number of LPs that need to be solved. This can best be understood when Equation (6.6) is viewed as a form of dynamic programming: Instead of pruning the cross-sum $\oplus_{z \in Z} \mathcal{V}^{a,z}$ directly, IP breaks it down by recursively computing $\mathbb{PR}(\oplus_{i=2}^{k} \mathcal{V}^{a,z_i})$ first, and then prune the cross-sum

$$\mathcal{V}^{a,z_1} \oplus \mathbb{PR}(\oplus_{i=2}^{k} \mathcal{V}^{a,z_i}).$$

Because the size of $\mathbb{PR}(\oplus_{i=2}^{k} \mathcal{V}^{a,z_i})$ is potentially much smaller than $\prod_{i=2}^{k} |\mathcal{V}^{a,z_i}|$, the number of LPs needed to prune $\oplus_{z \in Z} \mathcal{V}^{a,z}$ is reduced from $\prod_z |\mathcal{V}^{a,z}|$ to

$$|\mathcal{V}^{a,z_1}| \times |\mathbb{PR}(\oplus_{i=2}^{k} \mathcal{V}^{a,z_k})|.$$

Note that this argument applies equally to the recursive step $\mathbb{PR}(\oplus_{i=2}^{k} \mathcal{V}^{a,z_i})$. In general, the total number of LPs used by IP and its variants in computing Equation (6.1) is asymptotically $|\mathcal{V}^a| \sum_z |\mathcal{V}^{a,z}|$ [26]. Note that for typical POMDPs, $|\mathcal{V}^a|$ is usually smaller than, but nevertheless on the same order as, $\prod_z |\mathcal{V}_z^a|$.

Another bottleneck in computing Equation (6.1) is caused by the number of constraints in each of the linear programs that needs to be solved. From the procedure LP-DOMINATE in Table 6.1, each linear program solved has $|\mathcal{D}|$ inequality constraints, where $|\mathcal{D}|$ eventually approaches $|\mathcal{V}^a|$ when computing Equation (6.1). Again, this is exponential in the size of the previous value function. Although IP can effectively reduce the number of linear programs that need to be solved, it does not address the issue of the number of constraints. As a result, when using IP to solve POMDPs, especially those with a large number of observations, the large number of linear programs is usually not the first obstacle that we encounter. Instead, what we usually observe is that the program gets stuck solving one of the linear programs, because it has too large a number of constraints. Our main contribution is to show how the number of constraints can be reduced dramatically without affecting the solution quality, while at the same time maintaining the same number of linear programs as IP.

## 6.2 Witness region

Recall that the value function of a POMDP is piece-wise linear and convex (PWLC), and there is a unique and minimal vector representation for a PWLC function. Figure 6.1 shows an example of a value function minimally represented by three vectors. In such a representation, each vector $u \in \mathcal{U}$ defines a *witness region* $\mathcal{B}^u_{\mathcal{U}}$ over which $u$ dominates all other vectors in $\mathcal{U}$ [71]:

$$\mathcal{B}^u_{\mathcal{U}} = \{b|b \cdot (u - u') > 0, \forall u' \in \mathcal{U} - \{u\}\}. \tag{6.7}$$

For simplicity of notation, we use $\mathcal{B}_{\mathcal{U}}$ to refer to a belief region defined by some vector in $\mathcal{U}$, when the specific vector is irrelevant or understood from the context. We also use $\tilde{\mathcal{B}}$ to refer to some region when the vector and vector set are irrelevant or understood from the context.

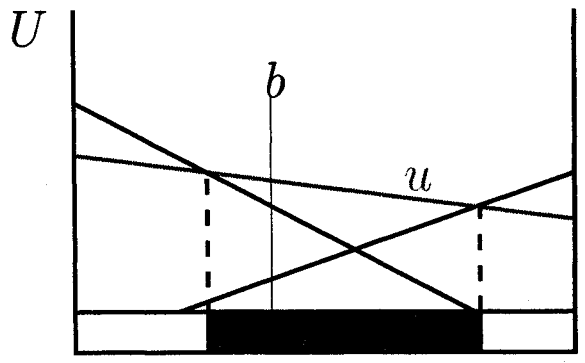


**Figure 6.1.** Witness region

Note that each inequality in Equation (6.7) can be represented by a vector, $(u - u')$, over the state space. We call the inequality associated with such a vector a *region constraint*, and use the notation $\mathbb{L}(\mathcal{B}_{\mathcal{U}}^{u}) := \{(u - u')|u' \in \mathcal{U} - \{u\}\}$ to represent the set of region constraints defining $\mathcal{B}_{\mathcal{U}}^{u}$. Note that for any two regions $\mathcal{B}_{\mathcal{U}}^{u}$ and $\mathcal{B}_{\mathcal{W}}^{w}$,

$$\mathbb{L}(\mathcal{B}_{\mathcal{U}}^{u} \cap \mathcal{B}_{\mathcal{W}}^{w}) = \mathbb{L}(\mathcal{B}_{\mathcal{U}}^{u}) \cup \mathbb{L}(\mathcal{B}_{\mathcal{W}}^{w}). \tag{6.8}$$

For each value function, there is an associated set of witness-regions that represents a partition of the belief-state space. Throughout the dynamic programming process, the value function is always finite, giving us a finite number of regions as well. We call this representation of the belief-state space a region-based representation. Note that the region-based representation is always associated with a vector representation of the value function and no new data structure is required to represent it. Therefore the region-based representation is more of a change of perspective when looking at the value functions of a POMDP. As I will show in this chapter, this change in perspective brings a dramatic improvement to the algorithm.

## 6.3 Region-based cross-sum pruning

In this section, we show how the explicit region-based belief-state representation can be exploited to greatly increase the performance of the cross-sum pruning operation. To simplify the notation, we drop the $a$ and $z$ superscripts in the cross-sum pruning Equation (6.1), and refer to the computation as

$$\mathcal{V} = \mathbb{PR}(\oplus_{i=1}^{k} \mathcal{V}_i). \tag{6.9}$$

Furthermore, we omit specifying the range of $i$ when the range is from 1 to $k$.

Consider the cross-sum set $\mathcal{U} \oplus \mathcal{W}$, where $\mathcal{U}$ and $\mathcal{W}$ are assumed to be minimal. It has been observed that [26]:

**Theorem 2** *Let $u \in \mathcal{U}$ and $w \in \mathcal{W}$. Then $(u+w) \in \mathbb{PR}(\mathcal{U} \oplus \mathcal{W})$ if and only if $\mathcal{B}_{\mathcal{U}}^{u} \cap \mathcal{B}_{\mathcal{W}}^{w} \neq \phi$.*

**Proof** If $(u + w) \in \mathbb{PR}(\mathcal{U} \oplus \mathcal{W})$, then $\exists b \in \mathcal{B}$ such that $\forall (u' + w') \in \mathcal{U} \oplus \mathcal{W}$,

$$\text{if } (u+w) \neq (u'+w'), \text{ then } (u+w) \cdot b > (u'+w') \cdot b.$$

It follows that

$$\forall u' \neq u \in \mathcal{U},\ (u+w) \cdot b > (u'+w) \cdot b,$$

therefore $u \cdot b > u' \cdot b$ and $b \in \mathcal{B}_{\mathcal{U}}^{u}$. Similarly, $b \in \mathcal{B}_{\mathcal{W}}^{w}$. Thus $b \in \mathcal{B}_{\mathcal{U}}^{u} \cap \mathcal{B}_{\mathcal{W}}^{w}$ which implies $\mathcal{B}_{\mathcal{U}}^{u} \cap \mathcal{B}_{\mathcal{W}}^{w} \neq \phi$.

If $\mathcal{B}_{\mathcal{U}}^{u} \cap \mathcal{B}_{\mathcal{W}}^{w} \neq \phi$, then $\exists b \in \mathcal{B}_{\mathcal{U}}^{u} \cap \mathcal{B}_{\mathcal{W}}^{w}$, and so $b \in \mathcal{B}_{\mathcal{U}}^{u}$ and $b \in \mathcal{B}_{\mathcal{W}}^{w}$. Thus

$$\forall u' \neq u \in \mathcal{U},\ u \cdot b > u' \cdot b,$$

and

$$\forall w' \neq w \in \mathcal{W},\ u \cdot b > u' \cdot b.$$

```
procedure LP-INTERSECT($\mathcal{B}^{v_1}_{\mathcal{V}_1}, \mathcal{B}^{v_2}_{\mathcal{V}_2}, \ldots, \mathcal{B}^{v_k}_{\mathcal{V}_k}$)
1. construct the following linear program:
   variables: $b(s)\ \forall s \in S$
   maximize 0
   subject to the constraints
         $b \cdot (v_1 - v) > 0,\ \forall v \in \mathcal{V}_1 - \{v_1\}$
         $b \cdot (v_2 - v) > 0,\ \forall v \in \mathcal{V}_2 - \{v_2\}$
         $\vdots$
         $b \cdot (v_k - v) > 0,\ \forall v \in \mathcal{V}_k - \{v_k\}$
         $\sum_{s \in S} b(s) = 1$
2. if the linear program is feasible, return TRUE
3. else return FALSE
```

**Table 6.2.** Linear programming test for region intersection.

It follows that

$$\forall (u'' + w'') \neq (u + w) \in \mathcal{U} \oplus \mathcal{W},\ (u + w) \cdot b > (u'' + w'') \cdot b.$$

Thus $(u + w) \in \mathbb{PR}(\mathcal{U} \oplus \mathcal{W})$. ■

This conclusion can be easily generalized to the cross-sum of more than two sets:

**Corollary 1** *Let $\mathcal{V}_i, i \in [1, k]$ be sets of vectors. Let $v_i \in \mathcal{V}_i$. Then $\sum_{i=1}^{k} v_i \in \mathbb{PR}(\oplus_{i=1}^{k} \mathcal{V}_k)$ if and only if $\cap_{i=1}^{k} \mathcal{B}^{v_i} \neq \phi$.*

With the region-based representation for the belief state space, testing for region intersection can easily be accomplished by solving a linear program to test if the individual regions share a common belief point. The linear program is listed in the procedure LP-INTERSECT in Table 6.2. We call this linear program the intersection LP. The constraints of the intersection LP are simply the combination of the region constraints of each region, plus the simplex constraint of the belief state $b$. In other words, the size of the intersection LP is $\sum_{i=1}^{k} |\mathcal{V}_i| + 1$.

### 6.3.1 Intersection-based incremental pruning

Corollary 1 suggests that the problem of computing $\mathbb{PR}(\oplus_i \mathcal{V}_i)$ is equivalent to finding all intersecting regions defined by the different vector sets. We introduce the operator $\mathbb{I}(\{\mathcal{V}_i\})$ that takes as input a set of vector sets and produces a list of intersecting regions defined by those vector sets:

$$\mathbb{I}(\mathcal{V}_{i_1}, \ldots, \mathcal{V}_{i_t}) = \left\{(\mathcal{B}^{v_1}_{\mathcal{V}_{i_1}}, \ldots, \mathcal{B}^{v_t}_{\mathcal{V}_{i_t}}) | \cap_{j=1}^{t} \mathcal{B}^{v_j}_{\mathcal{V}_{i_j}} \neq \phi\right\}$$

Pruning of the cross-sums can then be expressed as

$$\mathbb{PR}(\oplus_i \mathcal{V}_i) = \left\{\sum_i v_i \middle| (\mathcal{B}^{v_1}_{\mathcal{V}_1}, \ldots, \mathcal{B}^{v_k}_{\mathcal{V}_k}) \in \mathbb{I}(\mathcal{V}_1, \ldots, \mathcal{V}_k)\right\} \tag{6.10}$$

A naive approach to compute $\mathbb{I}(\{\mathcal{V}_i\})$ is to enumerate all possible combinations of $\{\mathcal{B}_{\mathcal{V}_i}\}$, and test them for intersection using the intersection LP. This requires a total of $\prod_i |\mathcal{V}_i|$ LPs, but each LP has only $\sum_i |\mathcal{V}_i|$ constraints. A better approach would be to use an incremental process similar to IP: To compute $\mathbb{I}(\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_k)$, we test if

$$\text{LP-INTERSECT}(\mathcal{B}_{\mathcal{V}_1}, \mathcal{B}_{\mathcal{V}_2}, \ldots, \mathcal{B}_{\mathcal{V}_k})$$

is true for all combinations of $\mathcal{B}_{\mathcal{V}_1}$ and $(\mathcal{B}_{\mathcal{V}_2}, \ldots, \mathcal{B}_{\mathcal{V}_k})$, where

$$(\mathcal{B}_{\mathcal{V}_2}, \ldots, \mathcal{B}_{\mathcal{V}_k}) \in \mathbb{I}(\mathcal{V}_2, \ldots, \mathcal{V}_k),$$

and $\mathbb{I}(\mathcal{V}_2, \ldots, \mathcal{V}_k)$ is computed recursively in the same manner. The recursion stops at $\mathbb{I}(\mathcal{V}_{k-1}, \mathcal{V}_k)$, at which point the naive approach is used to compute the results. We call this algorithm for computing $\mathbb{I}$ and subsequently $\mathbb{PR}(\oplus_i \mathcal{V}_i)$ the *intersection-based incremental pruning* (IBIP).

Surprisingly, IBIP solves the exact same number of LPs as IP (and the RR variants). To see this, consider the top level of the recursion. The total number of combinations between $\mathcal{B}_{\mathcal{V}_1}$ and $(\mathcal{B}_{\mathcal{V}_2}, \ldots, \mathcal{B}_{\mathcal{V}_k})$, and hence the number of LPs needed, is

$$|\mathcal{V}_1| \times |\mathbb{I}(\mathcal{V}_2, \ldots, \mathcal{V}_k)| = |\mathcal{V}_1| \times |\mathbb{PR}(\oplus_{i=2}^{k} \mathcal{V}_i)|,$$

which is also the number of LPs needed at the top recursion of IP (see end of Section 3). Similarly the same numbers of LPs are solved at all recursive steps. It follows that the total numbers of LPs of the two approaches are the same: $|\mathcal{V}| \sum |\mathcal{V}_i|$.

However, all the LPs used in computing $\mathbb{I}$ have at most $\sum_{i=1}^{k} |\mathcal{V}_i|$ constraints. In particular, when computing $\mathbb{I}(\mathcal{V}_t, \ldots, \mathcal{V}_k)$, the number of constraints ranges between $\sum_{i=t+1}^{k} |\mathcal{V}_i|$ and $\sum_{i=t}^{k} |\mathcal{V}_i|$. Thus, to compute $\mathbb{PR}(\oplus_i \mathcal{V}_i)$, the IBIP algorithm requires the same number of LPs but with possibly an exponential reduction in the number of constraints compared to IP. The number of constraints does not depend on the size of the output set, as with IP and RR.

### 6.3.2 Region-based incremental pruning

In this section, we show how the number of constraints in IBIP can be further reduced. To make a direct comparison with the recursion in IBIP, we will start from $\mathcal{V}_k$: To compute $\mathbb{I}(\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_k)$, we first fix a region in $\mathcal{V}_k$, call it $\mathcal{B}_{\mathcal{V}_k}$, and find all the elements in $\mathbb{I}(\mathcal{V}_1, \ldots, \mathcal{V}_{k-1})$ that intersect with $\mathcal{B}_{\mathcal{V}_k}$. We repeat this for all the regions in $\mathcal{V}_k$.

To find all the regions in $\mathbb{I}(\mathcal{V}_1, \ldots, \mathcal{V}_{k-1})$ that intersect with $\mathcal{B}_{\mathcal{V}_k}$, we first find all regions in each $\mathcal{V}_i (1 \leq i \leq k-1)$ that intersect with $\mathcal{B}_{\mathcal{V}_k}$. Recall that each such region corresponds to a vector in the vector set, and the set of intersecting regions corresponds to some subset of vectors $\mathcal{V}_i' \subseteq \mathcal{V}_i$. $\mathcal{V}_i'$ can be precisely computed by the region-based pruning, $\mathcal{V}_i' = \mathbb{PR}(\mathcal{B}_{\mathcal{V}_k}, \mathcal{V}_i)$. Once all $\mathcal{V}_i'$ are computed, we then recursively compute $\mathbb{I}(\mathcal{V}_1', \ldots, \mathcal{V}_{k-1}')$, by fixing a $\mathcal{B}_{\mathcal{V}_{k-1}'}$ and then find all the elements in $\mathbb{I}(\mathcal{V}_1', \ldots, \mathcal{V}_{k-2}')$ that intersect $\mathcal{B}_{\mathcal{V}_k} \cap \mathcal{B}_{\mathcal{V}_{k-1}'}$.

```
procedure $\mathbb{I}^*(\tilde{\mathcal{B}}, \{\mathcal{V}_i | i \in [1,t]\})$
1. $\mathcal{K} \leftarrow \phi$
2. if $t = 1$
3.    $\mathcal{K} \leftarrow \{\mathcal{B}^v_{\mathcal{V}_1} | v \in \mathbb{PR}(\tilde{\mathcal{B}}, \mathcal{V}_1)\}$
4. else
5.    for each $v \in \mathcal{V}_t$
6.       $\mathcal{V}'_i \leftarrow \mathbb{PR}(\tilde{\mathcal{B}} \cap \mathcal{B}^v_{\mathcal{V}_t}, \mathcal{V}_i), i \in [1, t-1]$
7.       if $\exists i \in [1, t-1]$ such that $\mathcal{V}'_i = \phi$
8.          continue
9.       $\mathcal{D} \leftarrow \mathbb{I}^*(\tilde{\mathcal{B}} \cap \mathcal{B}^v_{\mathcal{V}_t}, \{\mathcal{V}'_i | i \in [1, t-1]\})$
10.      $\mathcal{K} \leftarrow \mathcal{K} \cup \{(\mathcal{B}_{\mathcal{V}_1}, ..., \mathcal{B}_{\mathcal{V}_{t-1}}, \mathcal{B}^v_{\mathcal{V}_t}) | (\mathcal{B}_{\mathcal{V}_1}, ..., \mathcal{B}_{\mathcal{V}_{t-1}}) \in \mathcal{D}\}$
11. return $\mathcal{K}$
procedure $\mathbb{I}(\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_k)$
12. return $\mathbb{I}^*(\mathcal{B}, \{\mathcal{V}_i | i \in [1,k]\})$
```

**Table 6.3.** Region-based pruning for computing $\mathbb{I}$.

Note that the $\mathbb{I}$ operator serves only as a conceptual place-holder in this process; all the computations are carried out using the region-based pruning operator.

Table 6.3 shows the algorithm that finds the set of intersecting regions using this process. We call the algorithm that computes $\mathbb{PR}(\oplus_i \mathcal{V}_i)$ using Table 6.3 and Equation (6.10) the *region-based incremental pruning* (RBIP) algorithm.

The main motivation for RBIP is to further reduce the number of constraints. As Table 6.3 shows, all pruning in RBIP is of the form $\mathbb{PR}(\tilde{\mathcal{B}}, \mathcal{V}_t)$. In line 3, the pruning corresponds to testing some $\mathcal{B}_{\mathcal{V}_1}$ with some $(\mathcal{B}_{\mathcal{V}_2}, \ldots, \mathcal{B}_{\mathcal{V}_k})$ for intersection in IBIP. The number of constraints in IBIP is from $\sum_{i=2}^k |\mathcal{V}_i|$ to $\sum_{i=1}^k |\mathcal{V}_i|$. The number of constraints in RBIP ranges between $\sum_{i=2}^k |\mathcal{V}^*_i|$ and $\sum_{i=1}^k |\mathcal{V}^*_i|$, where $\mathcal{V}^*_i$ is $\mathcal{V}_i$ pruned multiple times previously in line 6. Because of the region-based pruning, $|\mathcal{V}^*_i|$ could be much smaller than $|\mathcal{V}_i|$ and this is where the savings come from. The analysis of the pruning in line 6 follows similarly.

In addition to reducing the number of constraints, RBIP can also reduce the number of linear programs. In Table 6.3, during each recursive call to $\mathbb{I}^*()$, the sizes of input sets $\mathcal{V}'_i$ are already reduced by pruning. Thus each subsequent problem that $\mathbb{I}^*()$ solves can be progressively smaller. However, without assuming any special restriction on the geometric form of the value function, it is also possible that the region-based pruning in line 6 may not prune any vector at all. In this case there is no saving in the number of constraints as compared to IBIP. Further, if every region-based pruning falls in this worst-case scenario, the total number of LPs solved by RBIP will be $|Z||\mathcal{V}|\sum|\mathcal{V}_i|$, or $|Z|$ times that of IBIP. It remains an open question whether this happens in realistic POMDPs. In all the experiments we have performed so far, we observed substantial savings in terms of both the number of LPs and the number of constraints using RBIP. We present these results in the next section.

### 6.3.3 Empirical evaluation

In this section, we present experimental results comparing the performance of IBIP, RBIP, and the GIP algorithms which is the most efficient version of the original incremental pruning algorithm [25]. We used the POMDP code by Cassandra [27] as the basis of our implementation, and used his implementation of the GIP algorithm for comparison.

#### 6.3.3.1 Problems from the literature

We first tested the algorithms on a set of problems from the literature that are publicly available from [27]. These problems are listed in Table 6.4. "4x3" is a navigation problem in a 4x3 maze, originally described in [90]; "Shuttle" models the problem of transporting supplies between two space stations using a shuttle [29]; "Maze20" is an augmented navigation problem in a 5x4 maze [56]; "Iff" models the problem of determining if an approaching aircraft is a threat or not [26]. Cassandra's thesis contains a good overall description of all these problems [26].

Note that our algorithm addresses the exponential blow-up associated with the number of observations. With 2 observations, our algorithms are essentially the same as the GIP algorithm. Thus we chose problems that have more than 2 observations. In many such problems, the number of observations in the problem description is usually larger than the number of *actual* observations – observations that are possible in any given state. This *actual* number is the number of vector sets whose cross-sum needs to be pruned. We

| problem | $Z$ | $Z^*$ | T | algorithm | time | LP | C |
|---|---|---|---|---|---|---|---|
| 4x3 | 6 | 2 | 10 | GIP | 19.50 | 14.38 | 0.78 |
| | | | | IBIP | 11.91 | 14.38 | 0.78 |
| | | | | RBIP | 11.11 | 14.38 | 0.78 |
| Shuttle | 5 | 3 | 10 | GIP | 9.86 | 10.07 | 0.59 |
| | | | | IBIP | 7.58 | 10.07 | 0.56 |
| | | | | RBIP | 7.65 | 10.41 | 0.56 |
| Maze20 | 8 | 4 | 3 | GIP | >10hr | na | na |
| | | | | IBIP | 30649.68 | 3545.58 | 1559.97 |
| | | | | RBIP | 6540.66 | 914.09 | 238.43 |
| Iff | 22 | 20 | 2 | GIP | >10hr | na | na |
| | | | | IBIP | 400.11 | 608.41 | 22.00 |
| | | | | RBIP | 329.07 | 759.32 | 12.45 |

**Table 6.4.** Test results on problems from the literature. Times are shown in seconds except where noted.

show the total number of observations in each problem in column "$Z$", and the maximal number of actual observations for any action in column "$Z^*$". Column "T" is the number of iterations of DP ran to collect the data. Only data for the pruning of the cross-sums are shown, because that is the only part affected by our algorithms. The column under "time" is the time spent on the pruning of the cross-sums, in CPU seconds. The column "LP" is the total number of linear programs solved during the pruning, in $10^3$, and the column "C" is the total number of constraints in those linear programs, in $10^6$. A limit of 10 hours was set for all the algorithms in these tests, after which they were terminated.

For each problem, we list the results for the GIP algorithm in the top row, followed by IBIP and then RBIP. As we can see, for the 4x3 problem, where only 2 sets of vectors are cross-summed, there is no difference in the number of LPs and the number of constraints. Even so, the time used by GIP is slightly longer. We conjecture that this is due to the different LP formations used by the different algorithms.

For the problem "Shuttle", where there are 3 actual observations, our algorithms begin to show their advantage in terms of the number of constraints. For the two larger problems "Maze20" and "Iff", GIP cannot finish the cross-sum pruning for all actions within the 10

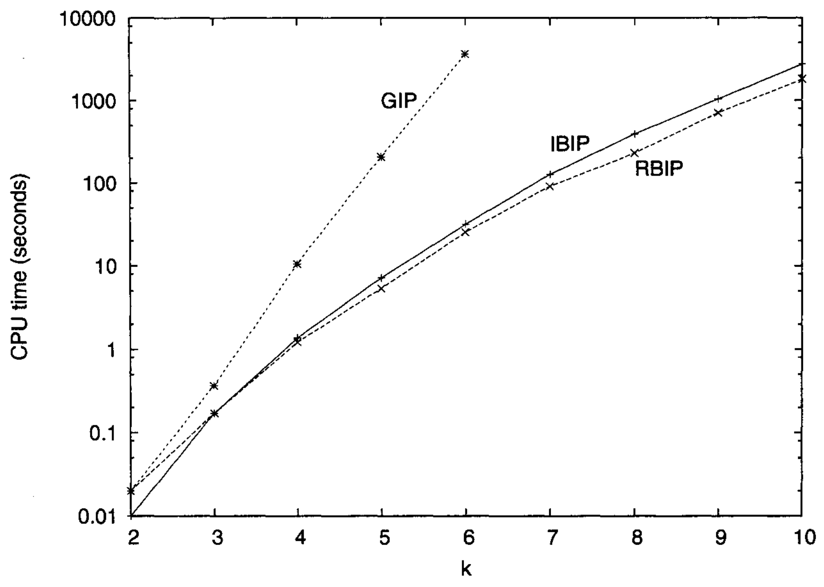


**Figure 6.2.** Timing result for problem set $(k, 10)$

hour limit. On "Maze20", GIP did not finish the pruning of a single set of the cross-sums involving 4 vector sets. On "Iff", GIP did finish the pruning of 2 of the 4 sets of cross-sums at the end of the 10-hour period (36572.52 seconds). Thus RBIP is at least 110 times faster than GIP on this problem. For the 2 sets processed, GIP solved $61.14 \times 10^3$ LPs, with a total of $28.96 \times 10^6$ constraints, which is already more than twice the total number of constraints solved by RBIP on all 4 sets.

On the two larger problems, RBIP uses significantly fewer constraints than IBIP. This is due to the region-based pruning. In "Shuttle" and "Iff", RBIP needs to solve slightly more LPs than IBIP, while in "Maze20", RBIP solves significantly fewer LPs than IBIP. In all cases, RBIP is at least as fast as IBIP, and in many cases a lot faster.

### 6.3.3.2 Artificial problems

Because of the lack of data collected for GIP on the larger problems from the literature, we present a special experiment to better demonstrate the scalability of our algorithms. We construct a set of random vector sets $\{\mathcal{V}_1, \ldots, \mathcal{V}_k\}$ and feed it directly to the algorithms to

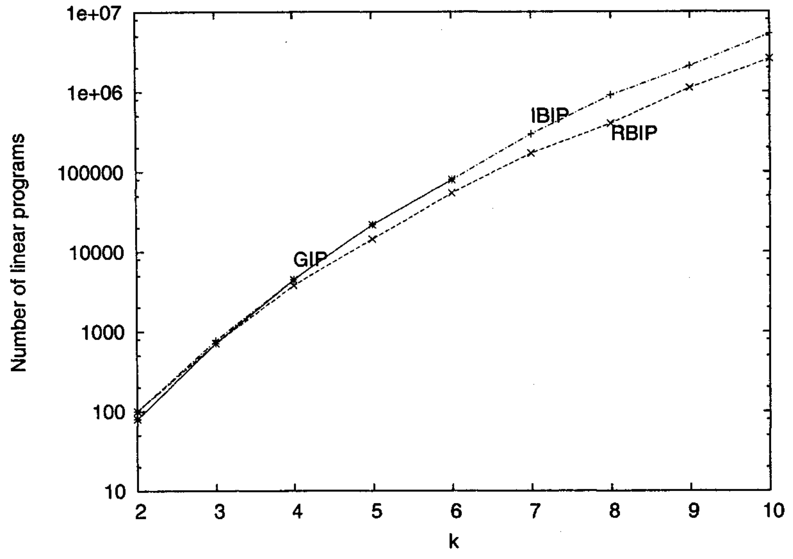


**Figure 6.3.** Number of linear programs solved for problem set $(k, 10)$

compute $\mathbb{PR}(\oplus_{i=1}^{k} \mathcal{V}_i)$. This way, we can easily vary the number of sets $k$ and the size of the input sets involved in the cross-sum.

Random vector sets, all with 10 states, are created as follows. The set $\mathcal{V}_i$ is initialized with a random vector, which is generated by drawing 10 numbers uniformly from $[-100.0, 100.0]$. Then, additional random vectors are generated and added to the set provided that they are not dominated. The procedure LP-DOMINATE$(v, \mathcal{V}_i)$ is used to determine if a new vector $v$ is dominated. This process is repeated until the number of vectors in $\mathcal{V}_i$ reaches $n$. A test problem is thus specified by the pair $(k, n)$.

It is hard to to determine whether vector sets created this way represent vector sets encountered in typical POMDPs. However we do note that it is easy to "reverse-engineer" a POMDP given an arbitrary set of vectors such that after one step of DP the exact same vectors are created. Hence each random test set corresponds to some POMDP.

We first illustrate the performance of the different algorithms on a problem set $(k, 10)$, where $k$ ranges from 2 to 10. Figure 6.2 plots the running time (in log scale) against $k$, the number of observations. It shows that both IBIP and RBIP outperforms GIP by orders of

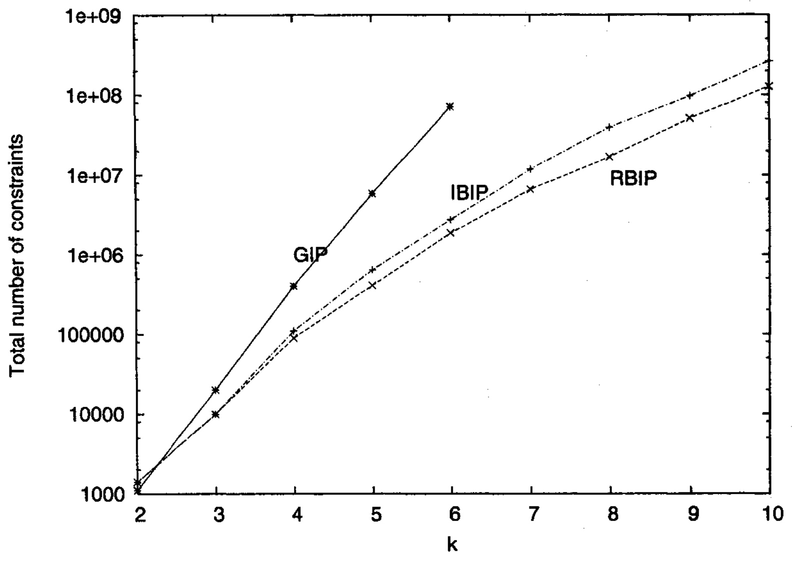


**Figure 6.4.** Total number of constraints for problem set $(k, 10)$

magnitude. For example, with 6 observations ($k = 6$), GIP needs about an hour (3665.81 seconds) to finish pruning the cross-sum, while IBIP and RBIP only use less than 1 minute (31.72 seconds and 25.66 seconds, respectively.) Beyond 6 observations, GIP cannot finish the pruning within 10 hours, while IBIP and RBIP finish the pruning for $k = 10$ in about 45 minutes (2738.04 seconds) and 30 minutes (1802.65 seconds), respectively. Also note that, except for $k = 2$ and $k = 3$, RBIP consistently outperforms IBIP by a sizeable margin.

Figure 6.3 shows the number of linear programs solved by the three algorithms, plotted in log scale. Note for GIP, only data for $k \leq 6$ are available, and except for $k = 2$, they are identical to data for IBIP. This matches the analysis we provide in Section 6.3.1. For the case of $k = 2$, GIP solved 80 linear programs, while both IBIP and RBIP solved 100. This is because GIP is specifically optimized for the case $k = 2$. For larger number of observations, the number of linear programs solved by RBIP is significantly less than that of IBIP. For $k = 10$, the number for RBIP is 1.11 millions, while the number of IBIP is 2.10 millions.

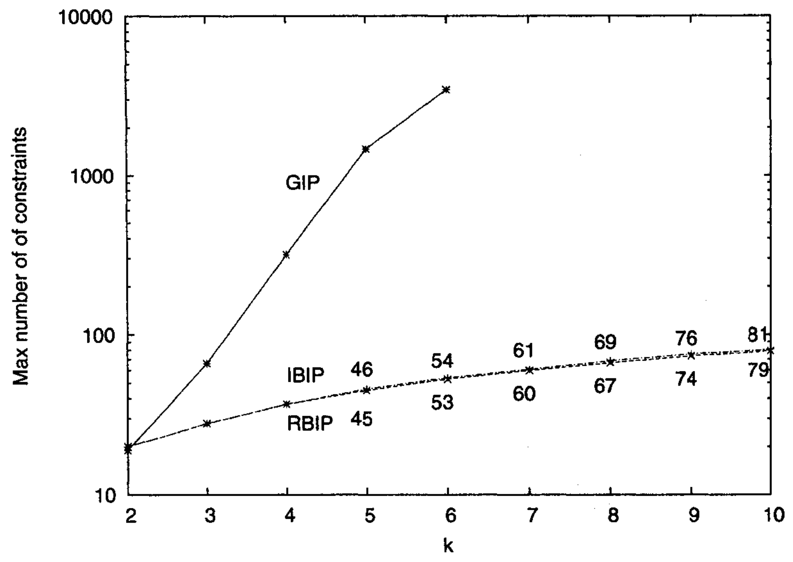


**Figure 6.5.** Max number of constraints for problem set $(k, 10)$

Figure 6.4 shows the total number of constraints in the linear programs solved by the three algorithms, and Figure 6.5 shows the maximal number of constraints in the linear programs, all plotted in log scale. Figure 6.5 shows clearly the drawback of GIP and the advantage of IBIP and RBIP. For GIP, the number of maximal constraints grows exponentially as the number of observations increases, while for IBIP and RBIP, the number only grows linearly. Although the asymptotic number of linear programs solved by all three algorithms are the same, IBIP and RBIP solve these linear programs exponentially faster than GIP. For large problems, GIP usually gets stuck solving just one of the linear programs. Note that curves for IBIP and RBIP are largely overlapping. The actual numbers, when they differ, are shown in the graph. As we can see, the maximal number of constraints in RBIP is always less than or equal to that in IBIP. This, combined with the fact that RBIP also solves fewer linear programs than IBIP, gives RBIP a clear advantage in performance over IBIP.

Finally, Table 6.5 presents data showing the speed-up factor of IBIP and RBIP over GIP on a range of problems $(k, n)$, where $k$ ranges from 2 to 6, and $n$ ranges from 15 to 35. The

| $k$ | $n$ | | | | |
|---|---|---|---|---|---|
| | 15 | 20 | 25 | 30 | 35 |
| 2 | 1.17 | 1.21 | 1.10 | 1.23 | 1.14 |
| | 1.17 | 1.21 | 1.14 | 1.23 | 1.13 |
| 3 | 2.98 | 4.08 | 4.53 | 4.78 | 4.22 |
| | 3.28 | 4.83 | 5.96 | 7.02 | 8.46 |
| 4 | 18.23 | 18.99 | 23.48 | 17.72 | 11.50 |
| | 23.35 | 30.78 | 31.98 | 53.85 | 39.27 |
| 5 | 78.28 | 81.77 | [216.39] | [476.60] | [680.51] |
| | 131.49 | 123.27 | [83.21] | [111.63] | [171.11] |
| 6 | [200.53] | [456.26] | [669.45] | [2009.74] | [3696.99] |
| | [143.10] | [176.36] | [238.38] | [433.55] | [655.01] |

**Table 6.5.** Speed-up factors compared to GIP on a range of problems $(k, n)$.

numbers in the table show the ratio between the time used by GIP for the pruning, and the time used by IBIP and RBIP. For each problem, data for IBIP is shown on the top row, and data for RBIP is shown on the bottom row. Numbers in brackets are actual running times (in CPU seconds) for the algorithms; no data is available from GIP to compute the speed-up factor on these problems, because GIP did not finish within the 10 hour limit. From this table, we can see that the performance gain depends mostly on the number of observations ($k$), and is relatively independent of the size of the state space ($n$).

## 6.4 Region-based maximization pruning

The maximization pruning presents yet another bottleneck in the DP process, since it needs to prune the union of the cross-sum value functions for all actions, and each cross-sum $\mathcal{V}^a$ can be exponential in the size of the previous value function $\mathcal{V}$. There has been relatively little work addressing maximization pruning, partly because the cross-sum pruning was the major bottle-neck that come before the maximization pruning step. With an exponential speedup to the cross-sum pruning, we are at a good position to address the maximization pruning step.

This section presents a simple algorithm for selecting constraints for the linear programs used in the maximization pruning stage. It exploits the locality structure of the belief state space. Again the region-based representation makes it possible to reason explicitly about these structures.

### 6.4.1 Projection pruning

Given the input value function $\mathcal{V}$, the linear programs in the projection pruning (Equation 6.2) have worst case number of constraints of $|\mathcal{V}^{a,z}|$. In the worst case, $|\mathcal{V}^{a,z}| = |\mathcal{V}|$. However, for many practical domains, $\mathcal{V}^{a,z}$ is usually much smaller than $\mathcal{V}$. In particular, a problem usually exhibits the following local structure:

- **Reachability**: from state $s$, only a limited number of states $s'$ can be reachable through action $a$.
- **Observability**: for observation $z$, there are only a limited number of states in which $z$ is observable after action $a$ is taken.

As a result, the belief update for a particular $(a, z)$ pair usually maps the whole belief space $\mathcal{B}$ into a small subset $T_a^z(\mathcal{B})$. Effectively, only values of $\mathcal{V}$ over this belief subset need to be backed up in the back projection in Equation 6.4. The number of vectors needed to represent $\mathcal{V}$ over this subset can be much smaller, and the projection pruning can in fact be seen as a way of finding the minimal subset of $\mathcal{V}$ that represents the same value function over $T_a^z(\mathcal{B})$. We will exploit this property in our algorithm, by shifting some of the pruning in the maximization stage to the projection stage of the next DP update.

### 6.4.2 Locality in belief space

Let

$$(v_1 + \cdots + v_k) \in (\mathcal{V}^{a,z_1} \oplus \cdots \oplus \mathcal{V}^{a,z_k})$$

refer to a vector in the cross-sum, implying $v_i \in \mathcal{V}^{a,z_i}$. From Corollary 1, $\sum_i v_i \in \mathcal{V}^a$ if and only if $\bigcap_i \mathcal{B}^{v_i}_{\mathcal{V}^{a,z_i}} \neq \phi$. Note that the witness region of $v = \sum_i v_i \in \mathcal{V}^a$ is exactly this intersection:

$$\mathcal{B}^{v}_{\mathcal{V}^a} = \bigcap_i \mathcal{B}^{v_i}_{\mathcal{V}^{a,z_i}}.$$

This gives us a way of relating the vectors in the output of the cross-sum stage, $\mathcal{V}^a$, to the regions defined by the vectors in the input vector sets $\{\mathcal{V}^{a,z_i}\}$. For each $v \in \mathcal{V}^a$, there is a corresponding list of vectors $\{v_1, v_2, \ldots, v_k\}$, where $v_i \in \mathcal{V}^{a,z_i}$, such that $v = \sum_i v_i$ and $\cap_i \mathcal{B}^{v_i}_{\mathcal{V}^{a,z_i}} \neq \phi$. We denote this list $parent(v)$.

**Proposition 1** *The witness region of $v$ is a subset of the witness region of any parent $v_i$:*

$$\mathcal{B}^{v}_{\mathcal{V}^a} \subseteq \mathcal{B}^{v_i}_{\mathcal{V}^{a,z_i}}; \tag{6.11}$$

Conversely, for each $v_i \in \mathcal{V}^{a,z_i}$, there is a corresponding lists of vectors $v^1, v^2, \ldots, v^m \in \mathcal{V}^a$, such that $v_i \in parent(v^j), \forall j$. We denote this list $child(v_i)$.

**Proposition 2** *The witness region of $v_i$ is the same as the union of its children's witness regions:*

$$\mathcal{B}^{v_i}_{\mathcal{V}^{a,z_i}} = \cup_j \mathcal{B}^{v^j}_{\mathcal{V}^a}. \tag{6.12}$$

The construction of the *parent* and *child* lists only requires some simple bookkeeping during the cross-sum stage. They will be the main building blocks of our algorithm.

### 6.4.3 Region-based maximization

Recall that in the maximization stage, the set $\mathcal{W} = \cup_a \mathcal{V}^a$ is pruned, where each $\mathcal{V}^a$ is obtained from the cross-sum pruning stage:

$$\mathcal{V}^a = \mathbb{PR}(\oplus_i \mathcal{V}^{a,z_i}).$$

Let us examine the process of pruning $\mathcal{W}$ using **procedure** $\mathbb{PR}$ in Table 6.1 (Page 81). In the `while` loop at line 14, an arbitrary vector $w \in \mathcal{W}$ is picked to compare with the current minimal set $\mathcal{D}$. As the size of $\mathcal{D}$ increases, the number of constraints in the linear programs approaches the size of the final result, $|\mathcal{V}'|$, leading to very large linear programs. However, to determine if some vector $w \in \mathcal{W}$ is dominated or not, we do not have to compare it with $\mathcal{D}$. Let $w \in \mathcal{V}^a$ and $v \in \mathcal{V}^{a'}$ for some $a$ and $a'$.

**Theorem 3** *If $a \neq a'$ and $\mathcal{B}^w_{\mathcal{V}^a} \cap \mathcal{B}^v_{\mathcal{V}^{a'}} = \phi$, then $w$ is dominated by $\mathcal{W}$ if and only if $w$ is dominated by $\mathcal{W} - v$.*

**Proof:** If $w$ is dominated by $\mathcal{W}$, that is, $\forall b \in \mathcal{B}, \exists u \in \mathcal{W}$ such that $w \neq u$ and $w \cdot b < u \cdot b$. If $\mathcal{W} - v$ does not dominate $w$, then $\exists b' \in \mathcal{B}^v_{\mathcal{V}^{a'}}$ such that $\forall v' \in \mathcal{W} - v, w \cdot b' > v' \cdot b'$. Since $a \neq a'$, $\forall v'' \neq w \in \mathcal{V}^a, w \cdot b' > v'' \cdot b'$ and therefore $b' \in \mathcal{B}^w_{\mathcal{V}^a}$. This contradicts the premise that $\mathcal{B}^w_{\mathcal{V}^a} \cap \mathcal{B}^v_{\mathcal{V}^{a'}} = \phi$. Therefore $w$ must be dominated by $\mathcal{W} - v$.

If $w$ is dominated by $\mathcal{W} - v$, then trivially it is also dominated by $\mathcal{W}$.■

**Corollary 2** *If $a = a'$ and $\mathcal{B}^w_{\mathcal{V}^a} \cap \mathcal{B}^v_{\mathcal{V}^{a'}-w} = \phi$, then $w$ is dominated by $\mathcal{W}$ if and only if $w$ is dominated by $\mathcal{W} - v$.*

Intuitively, to test dominance for $w$, we only need to compare it with vectors that have a witness region overlapping with the witness region of $w$. (Although we frame the theorem for the case of maximization pruning, it can be easily generalized to the pruning of any set of vectors.) However, finding these overlapping vectors in general can be just as hard as the original pruning problem, if not harder. So this result does not translate to a useful algorithm in general. Fortunately, for maximization pruning, the special setting in which the union of some previously cross-summed vectors are pruned allows us to perform a close approximation of this idea efficiently. We present a simple algorithm for doing so next.

### 6.4.4 Algorithm

We start by finding vectors in $\mathcal{V}^a - w$ that have a witness region overlapping with the witness region of $w$. From Equation 6.11, each vector $v_i \in parent(w)$ has a witness region

$\mathcal{B}^{v_i}_{\mathcal{V}^{a,z_i}}$ that fully covers the witness region of $w$. From Equation 6.12, each witness region $\mathcal{B}^{v_i}_{\mathcal{V}^{a,z_i}}$ is composed of witness regions of $child(v_i)$. Therefore the set

$$\mathcal{D}(w) = \{v | v \in child(v_i), v_i \in parent(w)\} \tag{6.13}$$

most likely contains vectors in $\mathcal{V}^a$ that have witness regions surrounding that of $w$, and their witness regions in the set $\mathcal{V}^a - w$ will overlap with the witness region of $w$.

Next we build a set of vectors in $\mathcal{V}^{a'}, a \neq a'$ that overlaps with the witness region of $w$. First, let $b(w)$ be the belief state that proved $w$ is not dominated in $\mathcal{V}^a$. This belief state is obtained from solving the linear program during the cross-sum pruning stage. We can find in the vector set $\mathcal{V}^{a'}$ a vector $v_{a'}$ that has a witness region containing $b(w)$, using procedure BEST in Table 6.1:

$$v_{a'} = \text{BEST}(b(w), \mathcal{V}^{a'}).$$

By construction, $v_{a'}$ and $w$ share at least a common belief state, $b(w)$. Now we use the same procedure as Equation 6.13 to build a set of vectors that covers the witness region of $v_{a'}$:

$$\mathcal{D}(v_{a'}) = \{v | v \in child(v_i), v_i \in parent(v_{a'})\}$$

Finally, we put together all these vectors:

$$\mathcal{D}' = \mathcal{D}(w) \cup \bigcup_{a' \neq a} \mathcal{D}(v_{a'}),$$

and use it to replace the set $\mathcal{D}$ at line 19 in Table 6.1 during maximization pruning. As a simple optimization, we replace $\mathcal{D}$ only when $|\mathcal{D}'| < |\mathcal{D}|$. The rest of the pruning algorithm remains the same.

Note that both $\mathcal{D}(w)$ and $\mathcal{D}(v_{a'})$ are incomplete. For $\mathcal{D}(w)$, it contains vectors that share a common parent with $w$, but there can be vectors that touch the boundary of the witness region of $w$ but don't share the same parent with it. For $\mathcal{D}(v_{a'})$, besides the same problem, the witness region of $v_{a'}$ may only partially overlap with that of $w$. Therefore the set $\mathcal{D}'$ constructed above does not guarantee that a dominated vector can be always detected. This does not affect the correctness of the dynamic programming algorithm, however, because the resulting value function still accurately represents the true value, albeit with extra useless vectors. These useless vectors will be included as the input to the next DP update step, in which their projections (Equation 6.4) will be removed during the projection pruning stage (Equation 6.2). At the cross-sum stage (Equation 6.1), the input vectors become the same as those produced by a regular DP algorithm that does not use our maximization pruning technique. Therefore the extra computation caused by the inaccurate pruning of our algorithm in the previous DP step happens at the projection pruning stage only.

As we will see in the next section, this extra computation is usually insignificant compared to the savings obtained from the maximization step. This may seem counterintuitive because the pruning of those undetected dominated vectors is not avoided, but merely delayed to the next step of the DP update. However, as explain earlier, the projection usually maps into a small region of the belief space, resulting in a larger number of vectors being pruned from the projection. As a result, the linear programs in the projection pruning are usually much smaller than the ones in the previous maximization pruning stage.

### 6.4.5 Empirical evaluation

In this section, we present experimental results on the maximization pruning algorithm implemented on top of the RBIP algorithm of Section 6.3.2. We call our algorithm RBIP-M, and compare its performance against RBIP. Note however that the region-based maximization algorithm only affects the maximization step of the standard DP update. There are many POMDP algorithms that use this standard DP update as a component. For example, [53, 101, 43, 102]. All these algorithms can be easily modified to incorporate the improvement offered by the region-based maximization algorithm.

| | | | | Time | | #LP proj | | Average #C proj | | #LP max | | Average #C max | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| problem | $\|S\|$ | $\|A\|$ | $\|Z\|$ | RBIP-M | RBIP | RBIP-M | RBIP | RBIP-M | RBIP | RBIP-M | RBIP | RBIP-M | RBIP |
| tiger | 2 | 3 | 2 | 20.28 | 20.39 | 7292 | 5446 | 19.81 | 19.11 | 4535 | 4527 | 11.26 | 19.04 |
| paint | 4 | 4 | 2 | 27.55 | 27.72 | 5033 | 2736 | 14.15 | 13.86 | 3325 | 2820 | 6.40 | 15.96 |
| shuttle | 8 | 3 | 5 | 681.39 | 608.43 | 58937 | 58533 | 28.49 | 29.64 | 84086 | 86500 | 200.36 | 219.38 |
| network | 7 | 4 | 2 | 1367.68 | 1992.16 | 128132 | 118749 | 25.24 | 25.47 | 207909 | 204708 | 103.31 | 283.63 |
| 4x3 | 11 | 4 | 7 | 5529.90 | 41567.91 | 11622 | 10765 | 58.31 | 63.32 | 31828 | 36155 | 636.25 | 6646.32 |

**Table 6.6.** Comparisons between RBIP-M and RBIP.

We test the algorithms on a set of benchmark problems from the literature. The number of states $|S|$, number of actions $|A|$ and number of observation states $|Z|$ of each problem are listed in Table 6.6. These problems are obtained from Cassandra's online repository [27]. All tests use a numerical precision of $10^{-6}$. The algorithm is considered converged when the error bound is less than 0.01, except for problem `4x3` (see below).

Our algorithm relies on two kinds of structures in a problem to perform well. First, the reachability and observability structure should be sparse so that the projection pruning can be much more efficient than the maximization pruning. The columns "Average #C proj" and "Average #C max" in Table 6.6 reflect this property. Second, the local structure of the belief regions defined by the vectors should allow neighboring relations among the regions to be adequately and efficiently captured by the *parent* and *child* lists. The adequacy is reflected by the "#LP proj" column, showing the extra number of linear programs that RBIP-M has to solve as a result of the undetected dominated vectors in the maximization stage. The efficiency is reflected by the reduction in the number of constraints in the maximization stage, shown in column "Average #C max".

For the problems `network` and `4x3`, RBIP-M is significantly faster than RBIP. (Coincidentally, these two problems are generally considered to be the harder problems in the literature.) This is because both structures are present in these problems. For example, in `4x3`, the average number of constraints in the projection pruning is about 60, much smaller than the number of constraints in the maximization stage. In addition, our algorithm is able to identify a much smaller set of vectors for use in the maximization linear programs (636.25 vs. 6646.32), while still effectively pruning most of the dominated vectors, resulting in only a small increase in the number of linear programs (from 10765 to 11622) solved during the projection stage. Combining these two factors gives our algorithm a great advantage. Note that for `4x3`, the data shown in Table 6.6 only represents the first 14 DP steps in both algorithms. At the end of the 14th iteration, RBIP already uses over 10 hours and is terminated. At this point RBIP-M is about 8 times faster than RBIP. The Bellman

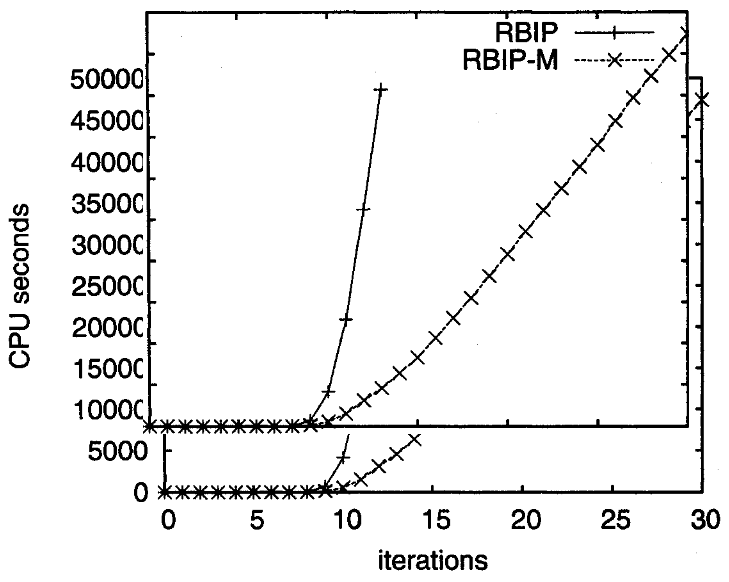


**Figure 6.6.** Running time comparison on problem 4x3.

residual at this point is 0.06. We continue to run RBIP-M on the problem for another 16 steps, reducing the Bellman residual to 0.03 using about the same amount of time required for the 14 steps of RBIP. The running time of these steps are plotted in Figure 6.6, and the average number of constraints in the maximization pruning is plotted in Figure 6.7. From these figures, we infer that the actual speedup of RBIP-M over RBIP on this problem can be much greater.

For the other three problems, one of the two structures is absent, leading to little performance improvement. In `tiger` and `paint`, the first structure is missing, as reflected by the number of constraints during the projection pruning being comparable to that during the maximization pruning. As a result, even though the maximization pruning deals with much smaller linear programs, the saving is offset by the extra cost incurred during the subsequent projection pruning. In the problem `shuttle`, the second structure is missing, as reflected by the fact that the number of constraints in RBIP-M (200.36) is only slightly smaller than that in RBIP (219.38). Therefore there is not much saving gained in the maximization pruning step and RBIP-M runs slower than RBIP in this case due to the extra linear programs solved in the projection stage.

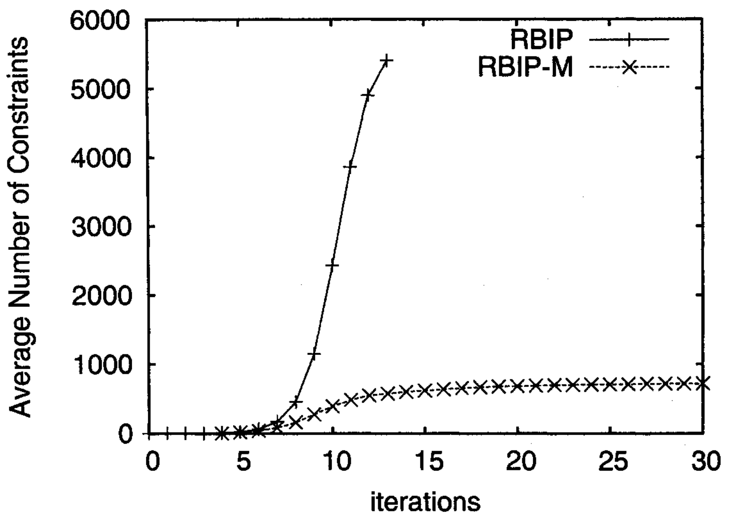


**Figure 6.7.** Average number of constraints in problem 4x3.

## 6.5 Summary

The main contribution of this chapter is the demonstration that there can be significant performance improvement to exact POMDP algorithms when the proper representations are used to exploit state abstraction in the belief space. The algorithms represent a major breakthrough in POMDP research over the past decade. This in turn leads to many new opportunities in POMDP research, which I will describe in detail in the next chapter.

# CHAPTER 7
# CONCLUSIONS AND FUTURE WORKS

## 7.1 Summary of contributions

Solving fully observable MDPs in general is unscalable because of the exponential growth of the state space. Solving partially observable MDPs in general is intractable because of the exponential growth of size of the vector representation of the value function over the uncountable belief state space. The main contribution of this thesis is in demonstrating the special structures that can be exploited to create more scalable and tractable algorithms for these models. There are two types of structures considered in this thesis:

- Domain specific state abstraction: This includes the discrete state abstraction considered in Chapter 3 and Chapter 5, and the continuous state abstraction considered in Chapter 4. There is strong evidence that many practical applications will contain those structure, and when they do, the representations and algorithms presented in this thesis are able to take advantage of the structures to solve the problems efficiently.

- Model specific state abstraction: This is the structure in the belief space of a POMDP described in Chapter 6. This is a stronger structure since it exists in all POMDPs, and the exponential speed-up by the IBIP algorithm (Section 6.3.1) over previous algorithms is guaranteed for all problem instances.

The two technical contributions of this thesis are:

1. New algorithms for fully observable MDPs that are expanding the limit on how large a problem can be realistically solved. The symbolic LAO* algorithm and symbolic

RTDP algorithm in Chapter 3, and the structured DP algorithm in Chapter 4 are all considered to be practical algorithms that can be applied to problems with billions of states.

2. Better understanding of the belief space structure for POMDPs. It was generally conceived that there could be no significant improvement to exact algorithms for POMDPs, and as a result, recent studies on POMDPs have mainly focused on approximation algorithms. We have shown that there are indeed significant (exponential) improvements possible. The new algorithms provide us with better tools to studies these complex models, and hopefully to discover new structures that can be exploited for further improvements.

## 7.2 Future directions

There are a number of exciting directions to build on the work presented in this thesis.

### 7.2.1 Heuristic search in the space of belief regions

Chapter 3 demonstrated the power of combining heuristic search techniques with state abstractions. The search component in that chapter can be easily extended to the continuous MDP model considered in Chapter 4. Here the search will happen in the space of rectangle regions in the continuous space. There has already been some work geared toward this direction [77].

More interestingly is to extend the search method to POMDPs using a region based representation. Search algorithm for POMDPs have been studied before [52, 57, 86]. In these previous work, the search happens in the space of unique belief states, that is, each node in the search tree is a single belief point. The size of the search tree grows exponentially in the number of actions and observations. However, many of these belief states may belong to the same witness region, and there is no need to distinguish them for the purpose of dynamic programming. Therefore searching in the space of belief regions could be more efficient than searching in the space of belief states.

### 7.2.2 Implementation of Sondik's policy iteration algorithm

The major difficulty with Sondik's policy iteration algorithm [96] is the canonical representation of the policy as a mapping from belief states to actions. In particular, Sondik's algorithm needs an explicit representation of the belief regions by enumerating the "edges" of the region. Hansen [52] circumvents this difficulty by representing the policy as a finite-state controller, avoiding totally the issue of representing the belief region. Although there are many advantages of such a representation, a major drawback is that the finite-state controller needs to store memory states that corresponds to dominated vectors. Thus, an equivalent policy in a canonical representation can be potentially much smaller, allowing more efficient policy evaluation and improvement.

Based on the region-based representation, it is possible to carry out Sondik's policy iteration without an explicit representation of the belief region. Instead, we can use the witness region implicitly defined by the set of vectors to represent a belief region. This avoids the complexity of having to deal with representing the "edges" in a high dimensional space.

As pointed out by Hansen, the class of policies that can be represented as a mapping from belief states to actions is different from the class of policies that can be represented as finite state controllers. The two classes only partially overlaps each other. Further, it can be shown that a policy is optimal only if it can be represented as a stationary mapping from belief to actions [11]. Therefore it is important to study Sondik's policy iteration algorithm and to understand what types of problems, if any, will lead to difficulties for Hansen's algorithm, and why.

### 7.2.3 Large Scale Distributed algorithms

The availability of large number of cheap computation platforms has prompted researchers with computational intensive applications to try to tap to this resource, by parallelizing their application. The best known example is probably the Seti@Home project [67], where any computers on the Internet can participate in decoding random radio waves received across the world, in the hope to find meaningful messages presumably sent by intelligent lifes from outside the earth.

There is an obvious way to parallelizing the DP update for fully observable, discrete state MDPs. In asynchronous DP, each state of the MDP can be updated in independent processors, and the whole value function will eventually converge [8, 9]. How to divide the computation is however a challenge in the face of very large state space. The state abstraction formed by the algorithms presented in Chapter 3 and 4 may provide an easy answer: to divide the computation across different abstract states.

The parallelization of POMDP algorithms is much harder, however. This is because of the vector representation that must be used to represent the value function: each vector spans across the whole belief state space. As reported in [88], a trivial parallelization leads to large overhead. On the other hand, the region-based cross-sum pruning algorithm (Section 6.3.2) can readily be parallelized, by allocating the pruning in each region to separate processes. Because the pruning within a particular region are completely independent to the other regions, there is virtually no overhead from the parallelization. If the parallelization is implemented in a shared memory architecture, there is no communication overhead as well. If the parallelization is implemented over a distributed architecture, then the only overhead is sending the set of vectors to be cross-summed to remote processes over the network. Compared to the time used for solving the linear programs, it can be expected that communication overhead will only be a small fraction of the overall computational time.

# APPENDIX

# BINARY SPACE PARTITION TREE

## A.1 Background

A general BSP (Binary Space Partition) tree is constructed as follow: We begin with an initial region of space $r$ (that supposedly encloses the space that we are interested into), and choose some hyperplane $h$ that intersects $r$. We then use $h$ to induce a binary partitioning on $r$, which gives two new regions: $r.ge = r \cap h^{>=}$ and $r.lt = r \cap h^{<}$, where $h^{>=}$ stands for the positive halfspace and $h^{<}$ the negative (open) halfspace. We then repeat this process recursively on $r.ge$ and $r.lt$.

Note that in some application one might choose to divide the space into three components, i.e., the open positive and negative halfspaces, and the dividing hyperplane itself. For our purpose, it is sufficient to include the hyperplane in one of the sub-partition. In addition, since we are only concerned about (hyper) rectangles, the hyperplanes that partitions the space are always parallel to some dimemsion. This enables us to simplify the intersection operation a lot.

Figure A.1 shows an example of a partition and its BSP representation. Note several things:

- The boundary of the whole region (in dashed-line) is not represented explicitly in the BSP tree;
- Some rectangle is divided up into smaller pieces, (e.g. the one with value 1.0);
- The BSP representation is not unique;

- In this example, the terminal cell contains a single float number. In general, it can contain arbitrary data, e.g., the discretized transition probabilities, the reward and value function represented by a list of alpha vectors, etc.

The algorithms and data structures described here are adapted from [82].

## A.2 Data structure

A node in a BSP tree represents a region, a cutting plane, and its two sub-regions. In the case of a terminal node, the cutting plane and the sub-regions are not defined. The data structure for our BSP tree is really simple, it contains the following fields:

- $dim$: the dimension along which the cutting plane is aligned
- $pos$: the position of the cutting plane in the cutting dimension
- $ge$ and $lt$: the two sub-regions induces by the cutting plane
- $data$: this is a pointer to the actual data in the case when the node is a terminal node.

We will call this data structure $BspNode$.

## A.3 Intersection algorithm

The basic idea of intersecting two BSPs defined over the same space is very simple: we will break down one BSP at its root, and insert its cutting plane into the other BSP. We then recursively process the two children of the root. So let's first introduct the procedure **Partition**, which takes as input a BSP tree $T$, and a cutting plane defined by the dimension $dim$ and the position $pos$, and output a new BSP tree that takes the input cutting plane as the root cutting plane. In the simpliest implementation, one can just construct a new BSP node with two copies of $T$ as its children. This will be a correct BSP tree that represent the answer. However, we can do better by simplifying each $T$ since they are now in a smaller space, one that's divided from it's original space by the input cutting plane. We

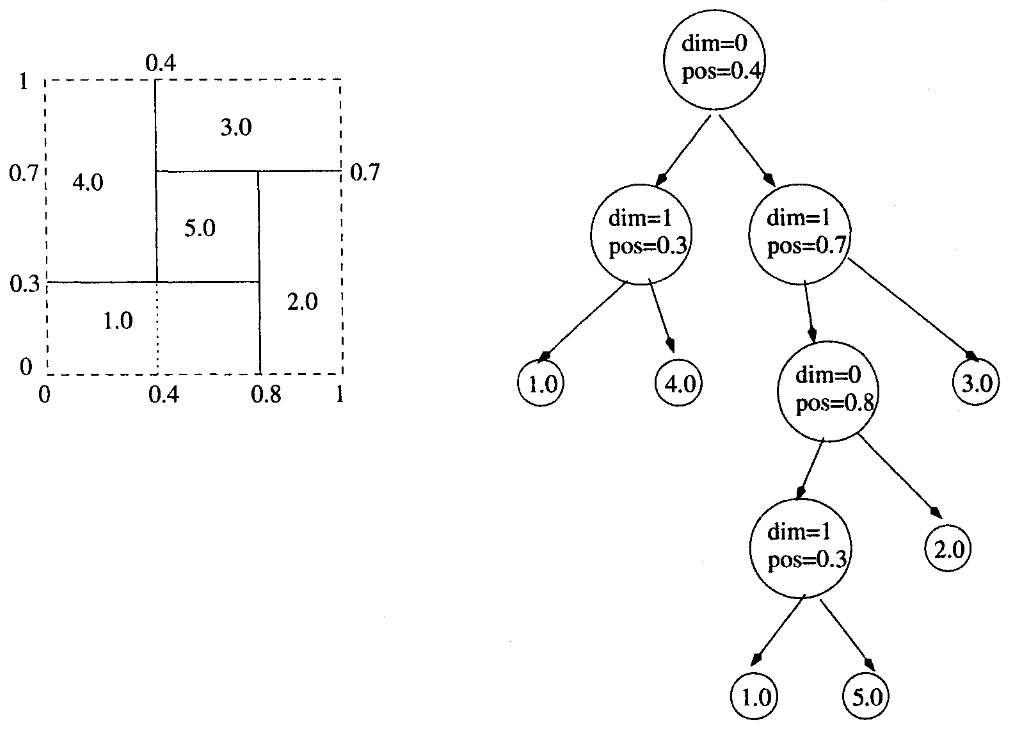


**Figure A.1.** BSP example

use the function **positiveHalf** and **negativeHalf** to represent such simplification process. They basically travers the tree and compare the region that each tree node represent with the halfspace defined by the cutting plane, and prune away any node that's outside of the halfplane. The partition is then implemented as follow:

```
Partition ($T, dim, pos$)
1. $R.dim = dim,\quad R.pos = pos$
2. $R.ge =$ positiveHalf($T.ge, dim, pos$)
3. $R.lt =$ negativeHalf($T.lt, dim, pos$)
4. return $R$
```

and the intersecting is implemented as follow:

```
Intersect ($T1, T2$)
1. if $T1$ or $T2$ is terminal node
```

```
2.         return IntersectTerminal($T1, T2$);
3. $P$ =Partition($T2, T1.dim, T1.pos$)
4. $R.d = T1.d$, $R.pos = T1.pos$
5. $R.ge$ =Intersect(T1.lt, P.lt)
6. $R.lt$ =Intersect(T1.ge, P.ge)
7. return $R$
```

Here the function **IntersectTerminal** performs the actual computation such as addition of maximization over the data stored at the terminal of the two BSP trees, which can be constants or piece-wise linear functions.

# BIBLIOGRAPHY

[1] Andre, David, Friedman, Nir, and Parr, Ronald. Generalized prioritized sweeping. In *Advances in Neural Information Processing Systems* (1998), vol. 10, MIT Press: Cambridge, MA.

[2] Astrom, Karl J. Optimal control of Markov decision processes with incomplete state estimation. *Journal of Mathematical Analysis and Applications 10* (1965), 174–205.

[3] Astrom, Karl. J. Optimal control of Markov processes with incomplete state information, II. *Journal of Mathematical Analysis and Applications 26* (1969), 403–406.

[4] Bahar, R. Iris, Frohm, Erica A., Gaona, Charles M., Hachtel, Gary D., Macii, Enrico, Pardo, Abelardo, and Somenzi, Fabio. Algebraic Decision Diagrams and Their Applications. In *IEEE /ACM International Conference on CAD* (Santa Clara, California, 1993), IEEE Computer Society Press, pp. 188–191.

[5] Barto, Andrew. G., Bradtke, Steven. J., and Singh, Satinder. P. Learning to act using real-time dynamic programming. *Artificial Intelligence 72* (1995), 81–138.

[6] Bellman, Richard E. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[7] Bertoli, Piergiorgio, Cimatti, Alessandro, and Roveri, Marco. Heuristic search + symbolic model checking = efficient conformant planning. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence* (2001), pp. 467–472.

[8] Bertsekas, Dmitri P. Distributed dynamic programming. *IEEE Transactions on Automatic Control 27* (1982), 610–616.

[9] Bertsekas, Dmitri P. Distributed asynchronous computation of fixed points. *Mathematical Programming 27* (1983), 107–120.

[10] Bertsekas, Dmitri P., and Tsitsiklis, John N. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996.

[11] Blackwell, David. Discounted dynamic programming. *Annals of Mathematical Statistics*, 36 (1965), 226–235.

[12] Blum, Avrim L., and Furst, Merrick L. Fast planning through planning graph analysis. *Artificial Intelligence*, 90 (1997), 281–300.

[13] Boutilier, Craig, Brafman, Ronen I., and Geib, Christopher. Structured reachability analysis for Markov decision processes. In *Proceedings of the 14th International Conference on Uncertainty in Artificial Intelligence* (1998), pp. 24–32.

[14] Boutilier, Craig, Dean, Thomas, and Hanks, Steve. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research* (1999), 1–94.

[15] Boutilier, Craig, Dearden, Richard, and Goldszmidt, Moises. Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence (IJCAI-95)* (Montreal, Canada, 1995), pp. 1104–1111.

[16] Boutilier, Craig, Dearden, Richard, and Goldszmidt, Moises. Stochastic dynamic programming with factored representations. *Artificial Intelligence 121* (2001).

[17] Boutilier, Craig, and Poole, David. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)* (Portland, OR, 1996), pp. 1168–1175.

[18] Boutilier, Craig, Reiter, Ray, and Price, Bob. Symbolic dynamic programming for first-order mdps. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)* (Seattle,WA, 2001), pp. 690–697.

[19] Boyan, Justin A. *Learning Evaluation Functions for Global Optimization.* PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1998.

[20] Boyan, Justin A., and Littman, Michael L. Exact solutions to time-dependent MDPs. In *NIPS 13*. 2000, pp. 1–7.

[21] Boyan, Justin A., and Moore, Andrew W. Generalization in reinforcement learning: Safely approximating the value function. In *NIPS 7* (1995), pp. 369–376.

[22] Brafman, Ronen I. A heuristic variable grid solution method for POMDPs. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)* (Providence, RI, 1997).

[23] Bresina, John, Dearden, Richard, Meuleau, Nicolas, Smith, David, and Washington, Rich. Planning under continuous time and resource uncertainty: A challenge for AI. In *Proc. of UAI-2002* (2002).

[24] Bryant, Randal E. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput. C-35*, 8 (Aug. 1986), 677–691.

[25] Cassandra, Anthony, Littman, Michael L., and Zhang, Nevin L. Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence* (1997), pp. 54–61.

[26] Cassandra, Anthony R. *Exact and Approximate Algorithms for Partially Observable Markov Decision Processes.* PhD thesis, Brown University, 1998.

[27] Cassandra, Anthony R., 1999. http://www.cs.brown.edu/research/ai/pomdp/.

[28] Cheng, H. *Algorithms for Partially Observable Markov Decision Processes.* PhD thesis, University of British Columbia, 1988.

[29] Chrisman, Lonnie. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the 10th National Conference on Artificial Intelligence* (1992), pp. 183–188.

[30] Cimatti, Alessandro, Roveri, Marco, and Bertoli, Piergiorgio. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence 159*, 1-2 (2004), 127–206.

[31] Cimatti, Alessandro, Roveri, Marco, and Traverso, Paolo. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *Proceedings of the 15th National Conference on Artificial Intelligence* (1998), pp. 875–881.

[32] Clarke, E.M., Emerson, E.A., and Sistla, A.P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems 8*, 2 (1986), 244– 263.

[33] Daniele, Marco, Traverso, Paolo, and Vardi, Moshe Y. Strong cyclic planning revisited. In *Proceedings of the 5th European Conference on Planning* (1999).

[34] Dean, Thomas, and Givan, Robert. Model minimization in markov decision processes. In *AAAI/IAAI* (1997), pp. 106–111.

[35] Dean, Thomas, Kaelbling, Leslie Pack, Kirman, Jak, and Nicholson, Ann. Planning under time constraints in stochastic domains. *Artificial Intelligence 76* (1995), 35–74.

[36] Dearden, Richard. Structured prioritised sweeping. In *The Eighteenth International Conference on Machine Learning* (2001).

[37] Dearden, Richard, and Boutilier, Craig. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence 89* (1997), 219–283.

[38] Dietterich, Thomas G., and Flann, Nicholas S. Explanation-based learning and reinforcement learning: A unified view. *Machine Learning 28* (1997), 169–214.

[39] Drake, A. *Observation of Markov Process Through a Noisy Channel.* PhD thesis, Electrical Engineering Department, MIT., 1962.

[40] Draper, Denise, Hanks, Steve, and Weld, Daniel. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)* (1994), pp. 31–36.

[41] Edelkamp, Stefan, and Reffel, Frank. OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)* (1998), pp. 81–92.

[42] Feng, Zhengzhu, Dearden, Richard, Meuleau, Nicolas, and Washington, Richard. Dynamic programming for structured continuous markov decision problems. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence (UAI-2004)* (2004).

[43] Feng, Zhengzhu, and Hansen, Eric A. Approximate planning for factored POMDPs. In *Proceedings of the 6th European Conference on Planning (ECP-01)* (2001).

[44] Feng, Zhengzhu, and Hansen, Eric A. Symbolic heuristic search for factored markov decision processes. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02)* (2002).

[45] Feng, Zhengzhu, and Hansen, Eric A. An approach to state aggregation for POMDPs. In *Proceedings of the 2004 AAAI workshop on learning and planning in Markov processes - Advances and challenges.* (San Jose, CA, 2004).

[46] Feng, Zhengzhu, Hansen, Eric A., and Zilberstein, Shlomo. Symbolic generalization for on-line planning. In *Proceedings of the 19th Conference on Uncertainty in Articial Intelligence (UAI-2003)* (2003).

[47] Feng, Zhengzhu, and Zilberstein, Shlomo. Region-based incremental pruning for POMDPs. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence (UAI-2004)* (2004).

[48] Feng, Zhengzhu, and Zilberstein, Shlomo. Efficient maximization in solving pomdps. In *Proceedings of the Twentieth of the Nineteenth National Conference on Artificial Intelligence (AAAI-05)* (Pittsburgh, PA, 2005).

[49] Friedman, J.H., Bentley, J.L., and Finkel, R.A. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Mathematical Software 3(3)* (1977), 209–226.

[50] Friedman, Nir, and Goldszmidt, Moises. Learning bayesian networks with local structure. In *Learning in Graphical Models*, Michael I. Jordan, Ed. MIT Press, 1999, pp. 421–460.

[51] Gordon, Geoffrey J. Stable function approximation in dynamic programming. In *Proc. 12th Intl. Conf. on Machine Learning* (1995), pp. 261–268.

[52] Hansen, Eric A. *Finite-memory control of partially observable systems.* PhD thesis, Department of Computer Science, University of Massachusetts at Amherst, 1998.

[53] Hansen, Eric A. Solving POMDPs by searching in policy space. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98)* (Madison, WI, 1998), pp. 211–219.

[54] Hansen, Eric A., and Zhou, Rong. Synthesis of hierarchical finite-state controllers for POMDPs. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling* (2003).

[55] Hansen, Eric A., and Zilberstein, Shlomo. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence 129* (2001), 35–62.

[56] Hauskrecht, Milos. Incremental methods for computing bounds in partially observable markov decision processes. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)* (Providence, RI, 1997).

[57] Hauskrecht, Milos. Value-function approximations for partially observable Markov decision processes. *Journal of Artificial Intelligence Research 13* (2000), 33–94.

[58] Hernandez, Natalia, and Mahadevan, Sridhar. Hierarchical memory-based reinforcement learning. In *Fifteenth International Conference on Neural Information Processing Systems* (Nov. 2000).

[59] Hoey, Jesse, St-Aubin, Robert, Hu, Alan, and Boutilier, Craig. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Articial Intelligence* (1999), pp. 279–288.

[60] Hölldobler, Steffen, and Skvortsova, Olga. A Logic-Based Approach to Dynamic Programming. In *Learning and Planning in Markov Processes–Advances and Challenges, Papers from the AAAI Workshop* (July 2004), AAAI Press, Menlo Park, California, pp. 31–36.

[61] Hong, Youpyo, Beerel, Peter A., Burch, Jerry R., and McMillan, Kenneth L. Safe BDD minimization using don't cares. In *DAC '97: Proceedings of the 34th annual conference on Design automation* (New York, NY, USA, 1997), ACM Press, pp. 208–213.

[62] Howard, Ronald A. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA, 1960.

[63] Howard, Ronald A., and Matheson, James E., Eds. *Readings on the Principles and Applications of Decision Analysis.* Strategic Decisions Group, Menlo Park, CA, 1984.

[64] Jensen, Rune M., Bryant, Randal E., and Veloso, Manuela M. Seta*: an efficient bdd-based heuristic search algorithm. In *Eighteenth national conference on Artificial intelligence* (Menlo Park, CA, USA, 2002), American Association for Artificial Intelligence, pp. 668–673.

[65] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: $10^{20}$ States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science* (Washington, D.C., 1990), IEEE Computer Society Press, pp. 1–33.

[66] Kaelbling, Leslie Pack, Littman, Michael L., and Cassandra, Anthony R. Planning and acting in partially observable stochastic domains. *Artificial Intelligence 101* (1998), 99–134.

[67] Korpela, E., Werthimer, D., Anderson, D., Cobb, J., and Leboisky, M. SETI@home-massively distributed computing for SETI. *Computing in Science & Engineering 3*, 1 (2001).

[68] Larsen, K.G., and Skou, A. Bisimulation through probabilistic testing. *Information and Computation 94*, 1 (1991), 1–28.

[69] Lind-Nielsen, Jorn. Buddy - a binary decision diagram package. *http://www.itu.dk/people/jln/* (1996).

[70] Littman, Michael L. The witness algorithm: Solving partially observable markov decision processes. Tech. Rep. CS-94-40, Brown University Department of Computer Science, 1994.

[71] Littman, Michael L., Cassandra, Anthony R., and Kaelbling, Leslie Pack. Efficient dynamic-programming updates in partially observable markov decision processes. Tech. Rep. CS-95-19, Brown University, Providence, RI, 1996.

[72] Littman, Michael L., Dean, Thomas L., and Kaelbling, Leslie Pack. On the complexity of solving Markov decision problems. In *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI–95)* (Montreal, Québec, Canada, 1995), pp. 394–402.

[73] Littman, Michael L., and Younes, Hakan L. The probabilistic planning track of the 2004 international planning competition. *http://www.cs.rutgers.edu/ mlittman/topics/ipc04-pt/* (2004).

[74] Lovejoy, William S. Computationally feasible bounds for partially observed markov decision processes. *Operations Research 39* (1991), 162–175.

[75] Lusena, Christopher, Mundhenk, Martin, and Goldsmith, Judy. Nonapproximability results for partially observable Markov decision processes. *Journal of Artificial Intelligence Research 14* (2001), 83–103.

[76] Madani, Omid, Hanks, Steve, and Condon, Anne. On the undecidability of probabilistic planning and related stochastic optimization problems. *Artificial Intelligence 147*, 1-2 (2003), 5–34.

[77] Mausam, Benazera, E., Brafman, R., Meuleau, N., and Hansen, Eric. A. Planning with continuous resources in stochastic domains. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)* (Edinburgh, Scotland, July 2005).

[78] Meuleau, Nicolas, Dearden, Richard, and Washington, Richard. Scaling up decision theoretic planning to planetary rover problems. In *Proceedings of the Workshop on Learning and Planning in Markov Processes: Advances and Challenges* (Menlo Park, CA, 2004), AAAI Press, pp. 66–71.

[79] Monahan, George E. A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science 28* (1982), 1–16.

[80] Moore, Andrew W., and Atkeson, Christopher G. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning 13* (1993), 103–130.

[81] Munos, Remi, and Moore, Andrew. Variable resolution discretization in optimal control. *Machine Learning 49*, 2-3 (2002), 291–323.

[82] Naylor, B., Amanatides, J., and Thibault, William. Merging BSP trees yields polyhedral set operations. *Computer Graphics (SIGGRAPH'90 Proceedings) 24(4)* (1990), 115–124.

[83] Papadimitriou, Christos H., and Tsitsiklis, John N. The complexity of Markov decision processes. *Mathematics of Operations Research 12*, 3 (Aug. 1987), 441–450.

[84] Parr, Ronald, and Russell, Stuart. Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (1995).

[85] Pearl, Judea. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, second ed. Morgan Kaufmann, 1988.

[86] Pineau, Joelle, Gordon, Geoff, and Thrun, Sebastian. Point-based value iteration: An anytime algorithm for POMDPs. In *Proceedings of the 2003 International joint conferene on artigicial intelligence (IJCAI-2003)* (2003).

[87] Puterman, Martin L. *Markov Decision Processes–Discrete stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, 1994.

[88] Pyeatt, Larry D., and Howe, Adele E. A parallel algorithm for POMDP solution. In *Proceedings of the 5th European Conference on Planning (ECP-99)* (September 1999), pp. 73–83.

[89] Roy, Nicholas, and Gordon, Geoff. Exponential family PCA for belief compression in POMDPs. In *Advances in Neural Information Processing 15* (2003).

[90] Russell, Stuart, and Norvig, Peter. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1995.

[91] Saul, L.K., and Jordan, M.I. Mixed memory markov models: Decomposing complex stochastic processes as mixture of simpler ones. *Machine Learning 37* (1999), 75–87.

[92] Smallwood, Richard D., and Sondik, Edward J. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research 21*, 5 (1973), 1071–1088.

[93] Somenzi, Fabio. CUDD: CU decision diagram package. *http://vlsi.colorado.edu/fabio/* (1998).

[94] Somenzi, Fabio. Binary decision diagrams. In *Calculational System Design*, M. Broy and R. Steinbruggen, Eds., vol. 173 of *NATO Science Series F: Computer and Systems Sciences*. IOS Press, 1999, pp. 303–366.

[95] Sondik, Edward. J. *The optimal control of partially observable Markov processes.* PhD thesis, Stanford University, 1971.

[96] Sondik, Edward. J. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research 26* (1978), 282–304.

[97] St-Aubin, Robert, Hoey, Jesse, and Boutilier, Craig. APRICODD: Approximate policy construction using decision diagrams. In *Proceedings of NIPS-2000* (2000).

[98] Sutton, Richard S., and Barto, Andrew G. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[99] Thrun, Sebastian. Monte Carlo POMDPs. In *Advances in Neural Information Processing (NIPS) 12* (2000), pp. 1064–1070.

[100] Yannakakis, Mihalis, and Lee, David. An efficient algorithm for minimizing real-time transition systems. In *5th International Conference Computer Aided Verification (CAV 93)* (1993).

[101] Zhang, Weihong, and Zhang, Nevin L. Solving informative partially observable markov decision processes. In *Proceedings of the 6th European conference on Planning (ECP-01)* (2001).

[102] Zhang, Weihong, and Zhang, Nevin L. Value iteration working with belief subset. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02)* (2002).

[103] Zhou, Rong, and Hansen, Eric A. An improved grid-based approximation algorithm for POMDPs. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence* (2001).