Ben-Gurion University of the Negev
Department of Computer Science

Learning and Solving Partially Observable Markov Decision Processes

Dissertation submitted in partial fulfillment
of the requirements for Ph.D. degree

by

Guy Shani

The research work for this dissertation has been carried out at
Ben-Gurion University of the Negev
under the supervision of Prof. Ronen I. Brafman and Prof. Solomon E. Shimony

July 2007

Subject: **Learning and Solving Partially Observable Markov Decision Processes**

This thesis is submitted as part of the requirements for the PhD. degree
Written by: **Guy Shani**
Advisor: **Dr. Ronen I. Brafman** Advisor: **Dr. Solomon E. Shimony**
Department: **Computer Science**
Faculty: **Natural Sciences**
Ben-Gurion University of the Negev

Author signature: _____ Date: _____

Advisor signature: _____ Date: _____

Advisor signature: _____ Date: _____

Dept. Committee Chairman signature: _____ Date: _____

Abstract

Partially Observable Markov Decision Processes (POMDPs) provide a rich representation for agents acting in a stochastic domain under partial observability. POMDPs optimally balance key properties such as the need for information and the sum of collected rewards. However, POMDPs are difficult to use for two reasons; first, it is difficult to obtain the environment dynamics and second, even given the environment dynamics, solving POMDPs optimally is intractable. This dissertation deals with both difficulties.

We begin with a number of methods for learning POMDPs. Methods for learning POMDPs are usually categorized as either model-free or model-based. We show how model-free methods fail to provide good policies as noise in the environment increases. We continue to suggest how to transform model-free into model-based methods, thus improving their solution. This transformation is first demonstrated in an offline process — after the model-free method has computed a policy, and then in an online setting — where a model of the environment is learned together with a policy through interactions with the environment.

The second part of the dissertation focuses on ways to solve predefined POMDPs. Point-based methods for computing value functions have shown a great potential for solving large scale POMDPs. We provide a number of new algorithms that outperform existing point-based methods. We first show how properly ordering the value function updates can greatly reduce the required number of updates. We then present a trial-based algorithm that outperforms all current point-based algorithms. Due to the success of point-based algorithms on large domains, a need arises for compact representations of the environment. We thoroughly investigate the use of Algebraic Decision Diagrams (ADDs) for representing system dynamics. We show how all operations required for point-based algorithms can be implemented efficiently using ADDs.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Uncertainty is a key feature in real world applications. In many cases agent actions have uncertain effects — a robot attempting to move forward may end up moving sideways due to engine malfunction or difficult terrain. In most cases such deviations from the expected outcome must also be detected by the agent. The robot must activate its sensors to obtain information as to its location. In many cases information about the environment is only partial and imprecise. The robot cameras or radars cannot provide information about hidden objects and, even for objects that can be seen, the information may be inaccurate due to sensor limitations.

Partially Observable Markov Decision Processes (POMDPs) (Howard, 1960; Sondik, 1971) provide a rich representation for such agents. POMDPs model aspects such as the stochastic effects of actions, incomplete information and noisy observations over the environment. POMDPs optimally balance the need to acquire information and the achievement of goals. We formally define and overview POMDPs, the required notation, and solution techniques in Section 2.2. Although POMDPs have been known for many decades, they are scarcely used in practice. This is mainly due to two major difficulties — obtaining the environment dynamics and solving the resulting model.

In most real world applications the environment dynamics, such as the probabilities of action outcomes and the accuracy of sensors should be learned. However, in a POMDP setting, it is likely that even the set of world states is initially unknown. A robot operating in an unknown environment, such as the Mars Rover, cannot know all the required environment details before it is deployed. In many cases the robot has access only to data arriving from its sensors, that usually provide only partial observation about the environment. Research has suggested two methods for dealing with this problem — model-based and model-free methods.

Model-based methods attempt to learn a complete description of the environment dynamics — the environment state, the action effect probabilities, and the probability of sensor outcomes. After learning the dynamics, the agent is able to employ a POMDP solver to obtain a policy that specifies the optimal actions. As learning the dynamics is a difficult task, model-free methods attempt to avoid this stage and compute a policy directly. Model-free methods are hence faster than model-based methods and in many cases compute a good policy.

We show, however, that when the environment provides noisy observations, model-free methods degrade rapidly. Sensor noise cannot be always avoided. While in many robotic applications the robot can be augmented using more accurate sensors, in other applications the noise cannot be reduced. Consider, for example, a diagnostic system, helping doctors identify the patients condition. When measuring the temperature of a patient the system might find that the patient does not have a fever. This does not mean that the patient does not have a flu, as even if the thermometer is accurate, a patient with a flu may or may not develop a fever. The observation noise in this case does not model the thermometer accuracy but rather the probability of having

a fever given a flu.

We suggest using a predefined sensor model — a definition of the probability of a possible observation given an environment feature. We show how model-free methods can be augmented using this sensor model to compute better policies in the presence of increasing sensor noise. However, the proper way to handle sensor noise is through a POMDP model. We thus move on to show how we can use the sensor model to learn a POMDP model from a number of model-free methods after their convergence. This two-stage learning scheme — first employ a model-free learning algorithm and only then learn a POMDP model – is undesirable. It is difficult to know when we should switch between stages, and the system has sub-optimal behavior for a long while, even though the collected data could be leveraged to produce better results. To remedy this shortcoming, we continue to present an online model-based algorithm, based on the well-known Utile Suffix Memory (USM — Section 3.1.6) (McCallum, 1996). Our online algorithm learns the environment state space, the environment dynamics, and a policy together. We show our online model-based approach to provide good policies in the presence of sensor noise.

After obtaining the system dynamics, the agent still needs to compute a policy. For a long time, policy computation was considered an intractable task. In the past few years, however, the emerging point-based approach (Pineau, Gordon, & Thrun, 2003) has been shown to be able to handle POMDP problems larger than ever before. This approximation method generates good policies rapidly. The second part of the dissertation is dedicated to improving point-based algorithms, designing new variations that can scale up to larger, more realistic applications.

In a POMDP environment, due to the inability to completely identify the environment state, the agent must maintain a belief state, specifying the probability of being in each world state. Most POMDP solvers compute a value function — a function assigning a value to each belief state. Point-based algorithms improve the value function by executing value function updates over specific belief states through a method known as a point-based backup.

We begin by analyzing the order by which value function updates are executed. We show that by properly arranging the order of backups, we can reduce the overall number of backups considerably. As a result the overall execution time of the algorithm is reduced. We explain how existing algorithms that maintain a set B of belief states and execute point-based backups over B can be accelerated by better organizing the order of backups. However, in larger domains, the maintenance of B becomes memory consuming. In such domains trial based algorithms scale up better. These algorithms execute trials, collecting a small number of belief states, compute point-based backups for these belief points only, and then discard the belief states and start a new trial. We propose a new trial-based algorithm that is among the fastest algorithms for solving POMDPs.

For larger domains with millions of states, the specification of environment dynamics must be compact. In the less complicated fully observable Markov Decision Process (MDP), structured representations that use Dynamic Bayesian Networks (DBNs) and Algebraic Decision Diagrams (ADDs) were successfully employed. Past attempts to use these representations for POMDPs did not produce considerable advantages, mainly due to the intractability of the POMDP solvers at that time. As solvers were only able to handle tiny problems, a compact representation did not seem an important issue. However, in view of the ability of point-based methods to handle millions of states, a compact representation is needed.

A straight forward extension of the ADD based operations used in MDPs does not scale up well. We fully explain how all required point-based operations can be implemented efficiently using ADDs. Our methods have the greatest impact when action effects are local, which is common in real world problems. We show our ADD based operations scale to larger domains unsolvable using ordinary 'flat' representation. Our methods are also superior to past attempts to use ADDs (Poupart, 2002).

The main contributions of this work are:

1. Contributions to the learning of environment dynamics
   (a) We demonstrate that some well-known algorithms for model-free learning perform poorly as noise in the environment increases.
   (b) We present model-free algorithms that better handle noisy sensors by learning using similar instances (Shani & Brafman, 2004).
   (c) We show how some model-free methods can be used to learn a POMDP model. We demonstrate that the resulting POMDP policy is superior to the policy of the original model-free method (?).
   (d) We suggest a method to incrementally build and maintain a POMDP model and its solution online (Shani, Brafman, & Shimony, 2005b).
2. Contributions to scaling up POMDP solvers
   (a) We propose a prioritization scheme to construct a good order of point-based backups, resulting in a considerable reduction in the overall number of backups (Shani, Brafman, & Shimony, 2005a).
   (b) We present a new trial-based and point-based algorithm that is currently able to solve the largest domains. Our method, FSVI, is currently the fastest method available for solving POMDPs and is also very easy to implement (Shani, Brafman, & Shimony, 2006).
   (c) We show how to implement any point-based algorithm over a compact representation that uses ADDs to model the system dynamics. This representation allows us to scale up to domains with millions of states.

The dissertation is structured as follows:

- Chapter 2 supplies necessary background for both parts of this dissertation. Specific background for each coming chapter will be presented at the beginning of the chapter.
- Chapter 3 presents our contributions in learning POMDP models:
  - We first present relevant work concerning learning under full and partial observability in Section 3.1. We overview model-free and model-based techniques and provide an introduction to McCallum's Utile Suffix Memory (USM) algorithm.
  - In Section 3.2 we show how the performance of some well known model-free RL algorithms degrades when sensors accuracy drops.
  - We proceed to present enhancements to McCallum's Instance-Based algorithms that better handle noisy sensors in Section 3.3.
  - Section 3.4 explains how the policy computed by some well known model-free algorithms can be used to instantiate a POMDP model of the environment.
  - Chapter 3.5 presents a method for incrementally building and maintaining a POMDP model based on the tree structure of the USM algorithm.
- In Chapter 4 we present a set of point-based algorithms
  - The chapter begins with an introduction to point-based algorithms for POMDPs in Section 4.1.

  - Section 4.2 demonstrates the influence of the order of point-based backups over the speed of convergence. We then present the Prioritized Value Iteration (PVI) algorithm.
  - We present the Forward Search Value Iteration (FSVI) — a fast, trial-based, point-based algorithm in Section 4.3.
  - We then explain how ADDs can be efficiently used in point-based algorithms, allowing us to scale to problems with millions of states (Section 4.4).

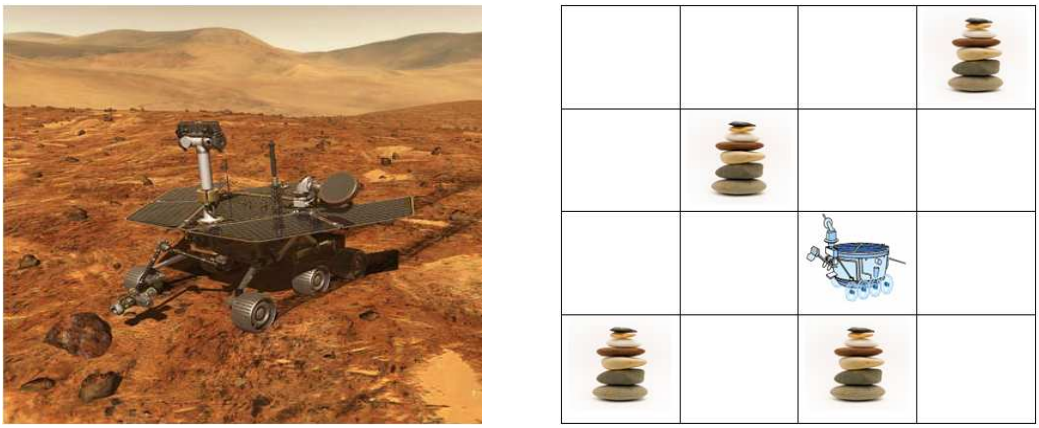
- Chapter 5 summarizes our contributions.

Chapter 2

Background

In this chapter we present some necessary background introducing the MDP and the POMDP models. We explain the necessary notations and overview well-known solution methods.

A dynamic system is called Markovian if the system transits to the next state, depending on the current state only. That is, if the system arrives at the same state twice, it will behave the same way (even though the behavior might be stochastic). Therefore, an agent operating in a Markovian system need not use memory at all. It is sufficient to observe the current state of the system in order to predict the system's future behavior.



(a) (b)

Figure 2.1: (a) A Mars Rover attempting to sample a rock (Courtesy of NASA/JPL-Caltech) and (b) a sketched model of the Mars Rover environment.

Consider the case of the Mars Rover robot — an autonomous agent operating on Mars. One of the tasks of the rover is to sample interesting rocks it observes around it[1]. The rover has to plan an optimal path that brings him to all interesting rocks, minimizing the required time and energy loss. To decide where to go next, the robot only needs to know the current environment state — its own location, the location of the rocks, which rocks have already been sampled, its own energy level, and so forth. The robot does not need to remember its history, such as his location in the previous time step, to decide on the next action.

In a controlled Markovian system the agent influences the environment through its actions, yet the effect of an action depends solely on the current state. To choose the next action optimally the agent needs only consider the current world state.

[1]This domain was formalized as the RockSample problem by Smith and Simmons (Smith & Simmons, 2004)

2.1 MDP

Autonomous agents operating in a Markovian environment can be modeled by a Markov Decision Process (MDP) (e.g., Howard, 1960; Bellman, 1962; Puterman, 1994; Kaelbling, Littman, & Moore, 1996). Formally, an MDP is a tuple $\langle S, A, tr, R \rangle$ when S is the set of states of the world, A is the set of actions available to the agent, tr is a transition function, and R is a reward function.

2.1.1 Definition

States A state $s \in S$ is a representation of all the relevant information in the world. In the Mars Rover problem the state should encapsulate the robot's $\langle x, y \rangle$ location, the location of the rocks, which rocks have been sampled, and the battery status. The robot does not need to know the color of the ground as this information is irrelevant for the completion of the task.

The size of the state space is a major concern in most applications. Since the solution of the MDP is polynomial in the number of states, it is necessary to keep the size of the state space small.

In this dissertation the state space is discrete and finite, but there are extensions to infinite and continuous state spaces.

Actions The agent modifies the world by executing actions. An action causes the agent to move from one state to another. For example, the Mars Rover actions might be — move forward, turn left, turn right, sample rock, and fill battery. It seems that some of these actions only modify the state of the robot, not the world, but as the robot is a part of the description of the world, modifying its state results in modifying the state of the world.

In an MDP framework the world state changes only after the agent executes an action.

Transitions When the robot tries to move in a certain direction it does not always succeed. Sometimes it might slip and move left or right. Sometimes the robot tries to turn left, but misses due to some engine inaccuracy or a sudden surge of electricity. The effects of actions are stochastic, leading to a stochastic transition between states. We write $tr(s, a, s')$ for the probability that an agent executing action a at state s will arrive at state s'. Since the model is Markov, the state transition depends only on the current state s and not on the history of the robot.

Rewards Rewards direct the agent towards desirable states of the world and keep it away from places it should not visit. The agent's goal is to maximize a stream of incoming rewards, or some function of it, such as the infinite horizon expected discounted reward $\sum_{t=0}^{\infty} \gamma^t r_t$. In our example the rover receives a reward each time it samples a rock that was not previously sampled, and should be punished (receive a negative reward) each time it samples a rock that was already sampled. Reward is typically not given simply for being in a certain state of the world, but rather for executing some action (such as drop package) in some state. The reward function is therefore written $R(s, a)$. It is also possible to consider stochastic reward functions but in this dissertation we restrict ourselves to deterministic rewards.

In many cases MDPs are used for planning under uncertainty. In such cases we usually assign a reward of 1 to the goal state and a reward of 0 elsewhere. Such a setting causes the agent to move as quickly as possible towards the goal. In other cases rewards may be given after every step.

It is also possible to model action cost as a negative reward. In the rover problem, for example, we may impose a cost for each robot action, related to the energy consumption induced by the action.

Usually the agent must optimize some function of the stream of rewards, such as the average reward per step or the infinite sum of discounted rewards.

2.1.2 Horizons and Discount Factors

An agent can operate under a finite horizon — a predefined number of time steps until the robot must finish its tasks. When the horizon is finite, the optimal action an agent can execute relies on the number of remaining time steps. For example, if the agent cannot fulfil the main objective within the remaining time steps, it may attempt a secondary objective that produces a smaller reward, but is faster to accomplish. Action selection in a finite horizon therefore depends not only on the current world state but also on the current time step.

In an infinite horizon the robot keeps executing the tasks forever. In such a case, therefore, there is no meaning to remaining time and the agent always has sufficient time to complete the tasks. In such a case, the optimal action depends only on the current state and there is no meaning to a time step.

In an infinite horizon case we usually introduce a discount factor $\gamma \in (0, 1)$ and the agent should maximize the infinite sum of discounted rewards:

$$\sum_{t=0}^{\inf} \gamma^t r_t \tag{2.1}$$

This infinite sum is bounded by $\frac{r_{max}}{1-\gamma}$ — where r_{max} is the maximal possible reward. This ensures the convergence of the algorithms below.

2.1.3 Solving MDPs — Policies and Value Functions

A stationary policy $\pi : S \rightarrow A$ is a mapping from states to actions. It is convenient to define the behavior of the agent through stationary policies. In the finite horizon MDP, a non-stationary policy is needed. Non-stationary policies π_t provide different actions over different time steps t for the same state. For the infinite horizon case, a stationary policy is sufficient. It was previously observed (Bellman, 1962) that an optimal deterministic policy π^* always exists for a discounted infinite horizon model. In this dissertation we restrict ourselves to the discounted infinite horizon case.

Value Iteration

An equivalent way to specify a policy is a value function. A value function $V : S \rightarrow R$ assigns a value for every state s denoting the expected discounted rewards that can be gathered from s. It is possible to define a policy given a value function:

$$\pi_V(s) = argmax_{a \in A}(R(s, a) + \gamma \sum_{s' \in S} tr(s, a, s')V(s')) \tag{2.2}$$

The optimal value function is unique and can be defined by:

$$V^*(s) = max_{a \in A}(R(s, a) + \gamma \sum_{s' \in S} tr(s, a, s')V^*(s')) \tag{2.3}$$

A policy π_{V^*}, corresponding to the optimal value function is an optimal policy.

Instead of defining a value for a state we can define a value for a state and an action $Q : S \times A \rightarrow R$ known as Q-values, where $Q(s, a)$ specifies the expected future discounted reward from executing action a at state s and acting optimally afterwards.

We can now define an algorithm for computing an MDP optimal solution for value iteration (Algorithm 1) (Bellman, 1962). The algorithm initializes the value function to be the maximal single step reward. In each iteration the value of a state is updated using the last estimation of the value of its successor states. We keep updating state values until they converge. The

Algorithm 1 Value Iteration

```
initialize V(s) = max_{a∈A} R(s, a)
while V does not converge do
  for all s ∈ S do
    for all a ∈ A do
      Q(s, a) ← R(s, a) + γ Σ_{s'∈S} tr(s, a, s') V(s')
    V(s) ← max_{a∈A} Q(s, a)
π(s) ← argmax_{a∈A} Q(s, a)
```

stopping criteria for value iteration can be used to create approximate solutions. A typical choice is to stop the algorithm when the maximal difference between two successive value functions (known as the Bellman residual) drops below ϵ. Value iteration is flexible in that it can be executed asynchronously or in some arbitrary order of assignments for the Q-value and will converge as long as the value of each $Q(s, a)$ gets updated infinitely often. Updating the Q-function based on the transition function is not necessary. We can replace the Q-function update with:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma max_{a' \in A} Q(s', a') - Q(s, a)) \tag{2.4}$$

As long as the Q-value for each s and a is updated infinitely often, s' is sampled from the $tr(s, a, s')$ distribution, $r = R(s, a)$ and the learning rate α is slowly decreased. Equation 2.4 is an important foundation for the model-free methods we shall describe later.

Other methods for solving MDPs are policy iteration that searches the space of policies instead of value functions, and linear programming. Linear programming is the only method theoretically known to converge to an optimal solution in polynomial time, but in practice, experiments on well known benchmarks suggest that it is slower than value iteration, policy iteration, and their various enhancements.

RTDP

In synchronous value iteration the entire state space is updated over each iteration. Asynchronous value iteration allows any arbitrary order of state updates. As a result, some states may be updated many times while other states are updated less often. A well known form of asynchronous value iteration is the trial-based approach. Simulation trials are executed, creating trajectories of states (for MDPs) or belief states (for POMDPs). Only the states in the trajectory are updated and then a new trial is executed. RTDP (Barto, Bradtke, & Singh, 1995) is a trial-based algorithm for MDPs (Algorithm 2).

An important aspect for the convergence speed of trial-based algorithms is a good heuristic for choosing the next action and state (lines 7 and 8). A good heuristic leads the trajectory towards important states over which the value function is updated. Unfocused heuristics may cause the algorithm to spend much time updating useless states.

Algorithm 2 RTDP

```
1: Initialize Q(s, a) ← 0
2: repeat
3:    s ← s_0
4:    while s is not a goal state do
5:       for each a ∈ A do
6:          Q(s, a) ← R(s, a) + Σ_{s'} tr(s, a', s')V(s')
7:       a ← arg max_{a'} Q(s, a')
8:       s ← sample from tr(s, a, ∗)
9: until the Q-function has converged
```

2.2 POMDP

In a Markov Decision Process the agent always has full knowledge about its whereabouts. For example, not only that such agents know their own current location, they are also aware of all other items in the world, regardless of whether they can see them or not. In practice this assumption is hard to implement. For example, robots have sensors such as cameras, proximity detectors, and compasses, providing incomplete data about the surroundings. A robot in a maze cannot usually know exactly its $\langle x, y \rangle$ coordinates. It observes walls around it and its direction and has to deduce from these data about its whereabouts. Moreover, sensors are inaccurate and return noisy output, causing more confusion as to the current state of the world.

In our Mars Rover scenario, consider the case where the rocks around it might contain an interesting mineral or not. The rover can activate a long range sensor to check whether the rock is interesting prior to approaching the rock. The accuracy of the long range sensors drops as the distance from the rock increases. The environment is still Markovian, but the rover has only partial information about it.

2.2.1 Definition

Formally, we assume that the agent is acting in a Markovian world, but it gains information over the state of the world through observations. The POMDP model (e.g., Poupart, 2002; Cassandra, Kaelbling, & Littman, 1994) for describing such agents is a tuple $\langle S, A, tr, R, \Omega, O, b_0 \rangle$ where:

- S, A, tr and R define an MDP (Section 2.1), known as the underlying MDP. This MDP describes the environment the agent operates within.
- Ω is a set of observations — usually the set of all possible sensor outputs.
- $O(a, s', o)$ is the probability that an agent will observe $o \in \Omega$ after executing a, reaching state s'. The O function models the sensor inaccuracy (sensor noise).

A POMDP is not Markovian with respect to the observed quantities, namely, the observations. The trivial method to make it Markovian is to maintain the complete history of actions and observations. Thus, it is possible to provide a policy as a mapping from histories to actions. However, there is a simpler way that does not require us to keep an infinite length history.

Instead of remembering the entire history, the agent can maintain a belief state — a vector of probabilities, where $b(s)$ is the probability that the agent is at state s. b_0 defines the initial belief, before the agent has executed an action or received an observation. When the agent is at belief state b, executing action a and observing o, it can calculate the new belief state

$b' = \tau(b, a, o)$:

$$
\begin{aligned}
b'(s) &= pr(s|a, o, b) && (2.5)\\
&= \frac{pr(o|s, a)pr(s|b, a)}{pr(o|b, a)} && (2.6)\\
&= \frac{O(a, s, o)\sum_{s' \in S} b(s')tr(s', a, s)}{pr(o|b, a)} && (2.7)\\
pr(o|b, a) &= \sum_{s \in S} b(s) \sum_{s' \in S} tr(s, a, s')O(a, s', o) && (2.8)
\end{aligned}
$$

A belief state is a sufficient summary of the history. In fact, many histories can map to the same belief state and thus, working in belief space is easier than computing a policy over histories.

Nevertheless, as some of the algorithms below operate in history space, we define here the notion of instances (?). An instance $T_t = < a_0, o_0, a_1, o_1, ..., a_t, o_t >$ is a prefix of some possible trial that was observed during interactions with the environments. Some of our algorithms operate using a set of instances. Such a set can be obtained by executing multiple trials over the environment, resetting the agent to an initial state each time a trial is finished.

2.2.2 Solving POMDPs — Exact Methods and Approximation Methods

Solving POMDPs is a difficult task (Papadimitriou & Tsitsiklis, 1987). Research has suggested a number of approximation techniques and some exact algorithms. We review most of these below, except for point-based methods which we shall discuss later.

MDP-based Approximations

As the underlying MDP (the MDP that describes the environment) is always easier to solve than a POMDP, research has suggested a number of approximation methods that rely on the solution (Q-function) of the underlying MDP.

The *most likely state* (MLS) method chooses the optimal action for the state with the highest probability in the current belief state. It does not consider the degree to which we are certain of the current state, nor the value of the optimal function. The *Voting* method addresses these concerns by selecting the action that has the highest probability mass in the belief state vector. The Q_{MDP} approximation (Cassandra et al., 1994) improves on Voting in that it takes the Q-values into consideration. In Q_{MDP} we select the action that maximizes $\sum_s Q(s, a)b(s)$, where $b(s)$ is the value of the belief vector for state s.

The main problem with MDP-based approximations is that they do not take into consideration actions that gather observations. As in an MDP, the current state is always known, actions such as sensor activation do not have any value and will never be selected by any of the above techniques. Nevertheless, Q_{MDP} is widely used to initialize a value function rapidly.

The Belief-Space MDP

The traditional way for solving POMDPs is through the belief space MDP — an MDP over the belief space of the POMDP:

$$R(b,a) = \sum_{s\in S} b(s)R(s,a) \tag{2.9}$$

$$tr(b,a,b) = \sum_{o\in\Omega} pr(b'|b,a,o)pr(o|b,a) \tag{2.10}$$

$$pr(b'|b,a,o) = \begin{cases} 1 & , \quad \tau(b,a,o) = b' \\ 0 & , \quad otherwise \end{cases} \tag{2.11}$$

$$pr(o|b,a) = \sum_{s\in S} b(s) \sum_{s'\in S} b(s)tr(s,a,s')O(a,s',o) \tag{2.12}$$

This reduction from a POMDP to the belief space MDP is exact, meaning that a policy computed for the belief space MDP has identical value in the original POMDP.

The value iteration equation that computes the next value function V_{t+1} given the current value function V_t:

$$V_{t+1}(b) = max_{a\in A}(\sum_{s\in S} b(s)R(s,a) + \gamma \sum_{o\in\Omega} pr(o|b,a)V_t(\tau(b,a,o))) \tag{2.13}$$

can be decomposed into:

$$V_{t+1}(b) = \max_{a\in A} V^a(b)| \tag{2.14}$$

$$V^a(b) = \sum_{o\in\Omega} V_o^a(b) \tag{2.15}$$

$$V_o^a(b) = \frac{\sum_{s\in S} b(s)R(s,a)}{|\Omega|} + \gamma pr(o|b,a)V_t(\tau(b,a,o)) \tag{2.16}$$

Although the belief space MDP has a continuous state space, its value function can be approximated arbitrarily closely using a set of vectors (hyper-planes) and is therefore piecewise linear (in the finite horizon case) and convex (Sondik, 1971). We thus use a set of $|S|$-dimensional vectors, known as α-vectors, where each vector corresponds to an action. The value iteration equations can be written in terms of sets of vectors, where we compute the next set of vectors $\mathcal{V}_{t+1}$ given the former set of vectors $\mathcal{V}_t$:

$$\mathcal{V}_{t+1} = \bigcup_{a\in A} \mathcal{V}^a(b)| \tag{2.17}$$

$$\mathcal{V}^a = \bigoplus_{o\in\Omega} \mathcal{V}_o^a(b) \tag{2.18}$$

$$\mathcal{V}_o^a = \{\beta_{\alpha,a,o}|\alpha \in \mathcal{V}_t\} \tag{2.19}$$

$$\beta_{\alpha,a,o}(s) = \frac{R(s,a)}{|\Omega|} + \gamma \sum_{s'\in S} b(s')tr(s',a,s)O(a,s,o) \tag{2.20}$$

where $A \oplus B = \{\alpha + \beta|\alpha \in A, \beta \in B\}$.

An action is assigned to each α-vector — the action that was used to generate it. All vectors in the $\mathcal{V}^a$ set are assigned the action a. The policy π_V of the value function V is immediately derivable using:

$$\pi_V(b) = \arg\max_{a:\alpha_a\in V} \alpha_a \cdot b \tag{2.21}$$

Where $\alpha \cdot b$ is the inner product operation:

$$\alpha \cdot b = \sum_s b(s)\alpha(s) \tag{2.22}$$

Zhang et al. (Cassandra, Littman, & Zhang, 1997) show that when computing $\mathcal{V}_{t+1}$, $\mathcal{V}^a$ and $\mathcal{V}^a_o$, the sets of vectors can be pruned to remove dominated elements, keeping the vector sets minimal. The resulting algorithm, using smart pruning techniques, can provide solutions to problems with small state spaces.

Finite State Controllers

A Finite State Controller (Hansen, 1998) (also known as a policy graph) is a finite state automaton, where states are labeled by actions, and edges are labeled with observations. When the agent is at a certain state in the FSC it executes the action associated with the state. The action triggers a change in the world resulting in an observation. Given the observation, the agent changes the internal state of the automata. Such controllers can be used to define a policy for a POMDP.

Hansen shows that given a POMDP model we can learn an FSC that captures an optimal policy, and considers this a form of policy iteration for POMDPs. In general, however, the FSC can grow such that the computation of an optimal controller becomes infeasible. FSCs, however, provide a good method for approximations techniques; by fixing the size of the controller, avoiding the problem above, the agent can find the best controller given the pre-defined size(Meuleau, Kim, Kaelbling, & Cassandra, 1999a; Braziunas & Boutilier, 2004; Poupart & Boutilier, 2004).

2.3 Factored Representations

Traditionally, the MDP/POMDP state space is defined by a set of states, which we call a flat representation. In many problems it is more natural to describe the environment using *state variables*. The state of a robot that must visit a predefined set of locations can be represented by state variables for its x and y coordinates and a binary variable for each location, stating whether it was already visited. When the state is defined through state variables we call the representation *factored*.

Formally, we define a set $X = \{X_1, X_2, ..., X_n\}$ of state variables. A state $s =< x_1, x_2, ..., x_n >$ is some assignment to the state variables. In many cases it is convenient to define the factored system dynamics, such as the transition, reward, and observation functions, through a Dynamic Bayesian Network (DBN) (Boutilier & Poole, 1996). We define a transition function DBN for each action and an observation function DBN for each action and observation. These DBNs capture the dependencies between states variables dynamics — which variables at time t affect variables at time $t+1$. As the Conditional Probability Tables (CPTs) of the DBN usually contain structure, it is usually more compact to represent them using decision trees or Algebraic Decision Diagrams (ADDs) (R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, & F. Somenzi, 1993).

Consider, for example, the network administration problem (Poupart & Boutilier, 2004), where a set of machines are connected through some network. Each machine may fail with some probability, which is increased when a neighboring machine is down. The task of the network administrator is to maximize the number of working machines, using restart operations over a single machine. The administrator must ping a machine to receive noisy status information (up/down).

This problem can be described using a state variable for each machine $X = \{M_1, ..., M_n\}$. When machine i is currently working (up) we set $M_i = T$.

A factored representation becomes compact when dependencies between state variables are limited. Formally, for each state variable X_i and action a we define X_i^a to be the minimal set of variables s.t. $pr(X_i'|a, X_i^a) = pr(X_i'|a, X)$. Smaller X_i^a sets result in a compact description of the $pr(X_i'|a, X_i^a)$ distribution. For example, if the network topology is a clique, then for each i and a $X_i^a = X$ and the problem does not factor well. If the topology is a ring, then $|X_i^a| = 3$ — machine i and its two neighbors – the problem has a compact factored representation. Poupart calls such independencies *context-specific* (Poupart, 2002).

2.4 Predictive State Representations

Predictive State Representations (PSRs) (Singh, Littman, & Sutton, 2001; Singh, James, & Rudary, 2004) present an alternative for POMDPs, shown to be superior in terms of the ability to efficiently model problems.

PSRs are designed to predict all possible future tests — sequences of instances — using some projection function over the given probability of observing a set of core tests. A prediction of a test is the probability of obtaining the observations in the test when executing the test's actions.

Predicting the probability of all possible future tests makes the selection of the next action easy — we can simply select the action that stochastically maximizes the sum of discounted rewards (or any other criteria) over all possible future tests. The learning algorithm for PSRs can be viewed as a three staged process:

- Discovering a good set of core tests (Rosencrantz, Gordon, & Thrun, 2004; James & Singh, 2004).
- Learning to predict every possible outcome using the core tests (Singh, Littman, Jong, Pardoe, & Stone, 2003).
- Learning a control policy using the core tests (James, Singh, & Littman, 2004).

Though PSRs were shown to provide models smaller than POMDPs that are at least as expressive, there has been little work on experimentally comparing PSRs with other model-based or with model-free algorithms. Rosencrantz et al. compared a Baum-Welch learned Hidden Markov Model with PSRs in terms of the ability to predict, demonstrating that given the model size, PSRs better predict future tests. They, however, did not show that a system that learns PSRs using their methods computes an optimal policy.

When comparing the size of McCallum's USM tree and a PSR, a good criterion would be the number of core tests compared with the number of leaves in the USM tree. In the deterministic case it is clear that even while we can set a core test for each environment state at the worst, we can still require a large number of histories to identify a single state. This is because the future and, hence, the tests are controlled, meaning that the agent can design a single test that will disambiguate two aliased states. The history, however, is uncontrolled and the agent cannot influence it.

Experiments shedding light on various factors of the learning process, comparing results from POMDP and PSR learning and solving, will contribute much to the understanding of both methods.

Chapter 3

Learning POMDP Models under Noisy Sensors

We begin this chapter with the model-based and model-free learning paradigms, and overview a number of popular algorithms in these classes. We give an extensive survey of McCallum's Utile Suffix Memory (USM) algorithm, which we shall later improve. We then show how noisy sensors cause a decrease in the performance of a number of well-known model-free methods, and explain why this happens.

We continue to augment USM using a sensor model to better handle noisy sensors. We then show how this sensor model can be used to learn a POMDP model using any model-free technique in a two stage scenario. Using a predefined sensor model to augment the learning of a POMDP is novel as far as we know. We finish by presenting an online POMDP learning algorithm that learns a POMDP and its policy incrementally.

3.1 Background

3.1.1 Reinforcement Learning Under Partial Observability

In this chapter we focus on agents that have no prior knowledge about the environment — the state space, the transition probabilities between states, the reward function, and so forth. We assume that the only information the agent has prior to the learning phase concerns its own abilities — the actions it can execute and the observations it receives. This setting is realistic in the case of a robot that initially knows which motors it has and which sensors it can operate. The agent must therefore learn to act through interactions with the environment.

This setting is usually known as the problem of reinforcement learning (RL) where an agent learns to operate in an environment through a sequence of interactions. Reinforcement learning was originally used for the fully observable MDP, but was also extended to the partially observable case.

3.1.2 The Model-Free and the Model-Based Learning Approaches

The standard method to model a reinforcement learning agent uses an MDP. To solve an MDP, through value iteration or policy iteration, the agent must know the system dynamics — the transition and reward functions. As such the agent must first learn the MDP model and only then solve it. This approach is known as the *Model-Based* approach.

Alternatively, it was observed that a policy can be learned without first learning all the system dynamics. This approach can be faster as it immediately computes a policy, without

first learning the MDP parameters.

We will discuss popular model-free and model-based approaches below, first in fully-observable environments, and then in partially-observable environments.

3.1.3 Model-based Methods in Markovian Domains

In a reinforcement learning setting, the transition and reward functions are initially unknown. The most straight-forward approach to acting in an unknown environment that can be modeled as an MDP is to learn the two functions by roaming the world and keeping statistics of the state transitions and rewards. Once a model has been learned, the agent can use a policy computation algorithm, such as value-iteration, to calculate an optimal policy. There are two major drawbacks to this method:

- Arbitrary division between learning and acting is undesirable. We usually prefer an agent that exhibits a learning curve, so that with each new experience it behaves a little better. One important advantage of model-free methods is that they keep exploring the world, and can therefore adapt to slow changes in the environment. Learning first and acting afterwards does not possess this attractive feature.

- It is unclear how the agent should explore the world. Random walks are undesirable as they might take a long time to learn the interesting parts of the world, while wasting time in places where no goal awaits. Random exploration might also cause an agent to repeatedly visit dangerous sections, receiving unneeded punishments.

The simple approach to model learning, known as the Certainty-Equivalent (e.g., Kaelbling et al., 1996) method, updates the model on each step (or every constant number of steps), solves it obtaining a new policy, and then acts by the new policy. While this approach addresses the issues above, it is extremely computationally inefficient and, hence, impractical.

Dyna

The Dyna system (Sutton, 1991) tries to combine model-free and model-based advantages. It incrementally learns both the Q-values and the model parameters, updating the policy and model simultaneously. Dyna keeps statistics for the state transitions and rewards and uses them to perform a part of the value iteration updates. As we noted before, value iteration does not require the agent to run sequential updates as long as each state-action pair is visited often enough.

Algorithm 3 Dyna(k)

initialize $Q(s,a) = 0$, $tr(s,a,s')$, $R(s,a)$
loop
  Choose action a using an ϵ-greedy policy on the Q-values of the current state s
  Execute a, observe reward r and new state s'
  Update: $tr(s,a,s')$ and $R(s,a)$ statistics
  Update: $Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s' \in S} tr(s,a,s') max_{a' \in A} Q(s',a')$
  for $i = 0$ to k **do**
    Choose s_i and a_i randomly
    Update: $Q(s_i,a_i) \leftarrow R(s_i,a_i) + \gamma \sum_{s' \in S} tr(s,a,s') max_{a' \in A} Q(s',a')$

Prioritized sweeping (Moore & Atkeson, 1993) improves Dyna by using a smarter selection of the k updates. Instead of choosing to update states at random, we prioritize the selection

of a pair $\langle s, a \rangle$ according to the variance in the latest updates to the successors states $\{s' \in S | tr(s, a, s') > 0\}$. Therefore, when an unexpected reward was observed, the system propagates this information to its predecessor states.

3.1.4 Model-free Methods in Markovian Domains

In order to use value or policy iteration (Algorithm 1), the agent needs a definition of an MDP model $\langle S, A, tr, R \rangle$. While it is reasonable to assume that S, the states of the world, and A, the available actions, are a part of the problem description, the state transition distribution and reward function might be not be known. Algorithms that learn to act in Markovian domains without a fully specified model of the environment (e.g., Sutton & Barto, 1998) are called model-free methods.

Model-free methods are also popular for solving MDPs with large domains as standard techniques on such domains require a full iteration over all the state space, which might be impractical.

Q-Learning

The most widely known method for learning in the absence of a model is Q-learning (Watkins & Dayan, 1992). Q-learning is popular as it is both simple and easy to implement. We define $Q^*(s, a)$ to be the expected discounted reward of executing action a in state s and behaving optimally afterwards. Since $V^*(s) = Q^*(s, a)$ we can write:

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} tr(s, a, s') \max_{a' \in A} Q^*(s', a') \tag{3.1}$$

In Q-learning (Algorithm 4), the agent executes an action a in state s arriving at state s' with reward r. It then applies Equation 2.4 to update the Q-value of the former state. If each state-action pair is updated infinitely and the learning parameter α decays slowly, the Q-function converges to Q^*.

Algorithm 4 Q-Learning

initialize $Q(s, a) = 0$
loop
  Choose action a using an ϵ-greedy policy on the Q-values of the current state s
  Execute a, observe reward r and new state s'
  Update: $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma max_{a' \in A} Q(s', a') - Q(s, a))$
  $s \leftarrow s'$

Choosing the next action should not always select the action with the maximal Q-value, to allow for some exploration. A standard exploration technique is e-greedy: choose the action with maximal Q-value with probability $1 - \epsilon$ and choose a random action with probability ϵ. In many cases stopping exploration at any stage is undesirable, since the environment might change and continuing to explore compensates for slow changes.

SARSA

A well-known variation of Q-learning is the SARSA algorithm (e.g., Sutton & Barto, 1998). SARSA is the same as Q-learning, except that the learning rule (Equation 2.4) is replaced with:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a)) \tag{3.2}$$

where s' is the actual next state and a' is the actual action that was selected in the next state. A well known problem with Q-learning performance in online tasks, is that its infinite exploration might cause it to take suboptimal actions, even though the optimal policy has already been discovered. Shutting off exploration is undesirable for several reasons:

- Infinite exploration helps the agent to handle worlds that slowly change over time.
- It is not clear when is it best to stop exploring. Splitting the online process to a learning and acting stage does not ensure convergence to an optimal solution.

SARSA is an on-policy learning algorithm. In updating the Q-values it takes into account the actual actions the policy selects at the next stage, not just the optimal action. It therefore also takes into consideration the exploration policy that is used.

Eligibility Traces

Eligibility traces (e.g., Singh & Sutton, 1996) introduce a different approach to the update step in the previous sections. We say that state-action pairs that were recently visited are more eligible for updating than those that were observed a long time ago. We add an eligibility trace to each pair $\langle s, a \rangle$, initiated when the agent executes action a in state s, decaying exponentially with parameter $0 \leq \lambda \leq 1$ afterwards, i.e., recently visited pairs have a higher trace. When the agent receives a reward or a penalty, the algorithm updates the Q-values of all the states based on their current trace. Therefore, all eligible state-action pairs take credit or blame for the event. It is possible to view the trace as a type of local memory, allowing agents to cope with partially observable domains with perceptual aliasing (Loch & Singh, 1998).

Algorithm 5 Sarsa(λ)

initialize $Q(s, a) = 0$
initialize $\eta_0(s, a) = 0$
$t \leftarrow 0$
loop
  Choose action a using an ϵ-greedy policy on the Q-values of the current state s_t
  Execute a_t, observe reward r_t and new state s_{t+1}
  for all state s and action a **do**
    $\eta_t(s, a) = \begin{cases} 1 & , \quad s = s_t, a = a_t \\ \gamma\lambda\eta_{t-1}(s, a) & , \quad otherwise \end{cases}$
    Update: $Q(s, a)_t \leftarrow Q_{t-1}(s, a) + \alpha\eta_t(s, a)(r + \gamma Q(s_t, a_t) - Q(s_{t-1}, a_{t-1}))$
  $t \leftarrow t + 1$

Some well-known variations to eligibility traces are the use of replacing traces for instantiating the trace of the current state-action pair, $\eta_t(s, a) = 1$, or accumulating traces where $\eta_t(s, a) = \eta_{t-1}(s, a) + 1$ is used. Singh et al. (Singh & Sutton, 1996) also recommended setting the traces for all other actions from the current state to 0.

$$\eta_t(s, a) = \begin{cases} 1 & , \quad s = s_t, a = a_t \\ 0 & , \quad s = s_t, a \neq a_t \\ \gamma\lambda\eta_{t-1}(s, a) & , \quad otherwise \end{cases} \tag{3.3}$$

Sarsa(λ) (Algorithm 5) is one of the common implementations of reinforcement learning with eligibility traces.

VAPS

Baird and Moore (Baird, 1999; Baird & Moore, 1998) developed an algorithm that can search both in the space of policies and the space of value functions — the Value And Policy Search (VAPS) algorithm. The VAPS algorithm uses gradient descent to minimize some error function on possible policies.

An instance represents a suffix of a sequence of interactions of an agent and the environment. We formally define an instance in a fully observable environment (following the definitions for a partial observability environment in (McCallum, 1996)) as $T_t = \langle T_{t-1}, a_t, r_t, s_t \rangle$, where the agent previously observed instance T_{t-1} and then executed action a_t, received reward r_t and arrived at state s_t. Error functions are defined for instances. We define the loss induced by a policy that generated the instance T_t as:

$$\varepsilon(T_t) = \sum_{i=0}^{t} e(T_t) \tag{3.4}$$

and we wish to minimize this loss over all possible instances:

$$B = \sum_{t=0}^{\infty} \sum_{T_t \in \mathcal{T}_t} pr(T_t)\varepsilon(T_t) \tag{3.5}$$

where $\mathcal{T}_t$ is the set of all possible instances of length t.

For example, the error measure for Q-learning can be:

$$e_{QL}(T_t) = \sum_{s \in S} tr(s_{t-1}, a_t, s)(r_{t-1} + \max_{a \in A} \gamma Q(s, a) - Q(s_{t-1}, a_{t-1}))^2 \tag{3.6}$$

and the error measure for SARSA, that averages the actions the policy might choose from state s_{t-1} instead of using the action that maximizes the Q-values:

$$e_{SARSA}(T_t) = \sum_{s \in S} tr(s_{t-1}, a_t, s) \sum_{a \in A} pr(a_t = a|s_{t-1})(r_{t-1} + \gamma Q(s, a) - Q(s_{t-1}, a_{t-1}))^2 \tag{3.7}$$

Baird and Moore also define an error measure for policies (rather than for Q-values) that simply maximizes the discounted utility:

$$e_{policy}(T_t) = b - \gamma^t r_t \tag{3.8}$$

where b is an constant used to make the error function values positive. It is also possible to linearly combine error measures:

$$e_{SARSA-policy}(T_t) = \beta e_{policy}(T_t) + (1 - \beta) e_{SARSA}(T_t) \tag{3.9}$$

As this criteria combines policy and value search it is known as VAPS.

Some stochastic policies can be represented as a vector of weights. For example, a policy that uses a Boltzman distribution on the Q-values to perform research can be represented as a matrix of weight when $w_{s,a} = Q(s, a)$. We can now define a measure called the exploration trace $ET_{s,a,t}$. The VAPS algorithm incrementally updates these two measures after instance T_t was observed:

$$\Delta ET_{s,a,t} = \frac{\partial}{\partial w_{s,a}} \ln pr(a_{t-1}|s_{t-1}) \tag{3.10}$$

$$\Delta w_{s,a} = -\alpha(\frac{\partial}{\partial w_{s,a}} e(T_t) + e(T_t) ET_{s,a,t} \tag{3.11}$$

Baird and Moore show that the gradients of the immediate error e with respect to weight $w_{s,a}$ is easy to calculate. The equations above approximate the stochastic gradient descent of the error measure B. Updates to the weights $w_{s,a}$ result in an update to the the stochastic policy represented by $pr(a|s)$.

Hierarchical approaches

Parr and Russell (1999) present the Hierarchy of Abstract Machines (HAM) architecture for reinforcement learning. A HAM is constructed of several layers of abstraction, each defined by a finite state machine whose states are either an action execution or a call to a lower level machine. Parr and Russell also show how to learn a HAM representing a policy without learning a model of the environment.

3.1.5 Perceptual Aliasing

In a fully observable MDP we assume that the agent might be unaware of the state transition and reward probabilities. The problem becomes harder in a partially observable domain when the agent is unaware of the state space at all. We consider cases where the agent is only aware of aspects of the problem that are a part of its structure — the actions it can execute, its sensors, their possible output signals, and their accuracy.

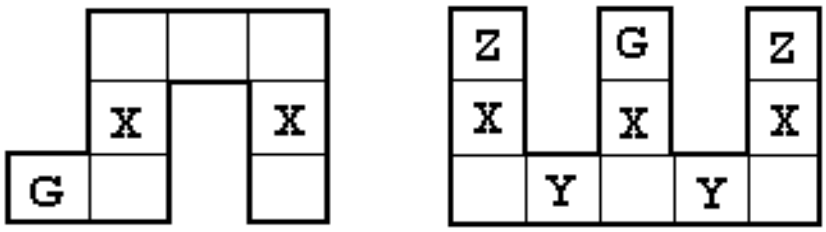
The trivial approach to cope with an unknown state space observed only through sensors is to use the observations instead of the unknown states, i.e., to assume that each observation corresponds to a single state. Using observations instead of states might lead to two opposite problems:

- The agent might have numerous sensors, each applicable to different aspects of the problem. The Mars Exploration Rover, for example, can sense many things about its surroundings, most of which are needed for the Mars research, but irrelevant for its navigational mission. Such agents need to learn to distinguish between relevant and irrelevant observations. We do not handle such problems in this paper.

- The observation space might be smaller than the state space, causing the agent to suffer from the problem of perceptual aliasing (Chrisman, 1992). Having numerous states that correspond to the same precept can be beneficial when the optimal action for each state is the same, as it diminishes the size of the state space. However, when two states generate the same observation, yet the agent needs to execute different actions in each state, we call the states perceptually aliased. The agent must learn to differentiate between them in order to compute an optimal policy, a problem we focus on below.

The problem of insufficient data leads to a phenomenon known as *perceptual aliasing* (Chrisman, 1992), where the same observation is obtained in distinct states where different actions should be performed. Some researchers (?, ?) define all states where the agent observes the same precept to be perceptually aliased, regardless of the optimal action for the state, but we choose to follow Chrisman's definitions. For example, in Figure 3.1(a) and 3.1(b), the states marked with X and Y are perceptually aliased. States marked with Z are not perceptually aliased as the optimal action for these states is identical. To compute an optimal policy, the agent must learn to disambiguate the perceptually aliased states.

The problem of perceptual aliasing is exacerbated when the agent's sensors are not deterministic. One such example is where walls can sometimes be reported where none exist, or where a wall is occasionally not detected. Though it is possible to augment some of the robot‘s sensors or add some additional sensors to filter out the noise, in general making observations



(a) (b)

Figure 3.1: Two maze domains. The agent should navigate to the goal state, marked by G.

about the world cannot be expected to be noise-free. Noisy sensors can also be used to model non-robotic problems such as expert systems. Consider, for example, a medical diagnostic system. A patient with an infection may or may not develop a fever. The observation of a fever is therefore stochastically dependent on whether the patient has an infection. The noisy output of the sensors in this example models the probability of having a fever in the presence of an infection, not the thermometer accuracy.

Reactive memoryless policies The simplest approach for handling an unknown state space is to use the observation space instead of the state space, and then apply some method applicable for the fully observable domains such as Q-learning or Sarsa to solve it, computing $Q(o, a)$ rather than $Q(s, a)$. These policies are known as memoryless or reactive, as they react to the latest observation solely without keeping track of past events. (Littman, 1994) provided theoretical limitations for memoryless policies. Littman also presented a branch-and-bound algorithm for computing optimal memoryless policies and examined its performance on a number of examples. The branch-and-bound algorithm relied on the existence of a POMDP model, making it model-based, rather than model-free.

(Whitehead & Ballard, 1991) suggested, in their Lion algorithm, to decrease the utility of states that are perceptually aliased so that the agent will try to avoid these states if possible. While this approach can be effective in some domains, it is generally required to be able to somehow visit perceptually aliased states.

(Jaakkola, Singh, & Jordan, 1995) showed that a deterministic reactive policy can perform arbitrarily worse than stochastic policies. Their work was continued by (Williams & Singh, 1998) who implemented an online version of the stochastic algorithm and tested it. They showed their algorithm to converge on a problem, but with inferior results (about 40% of the average reward per step) to Parr's model-based approximation techniques (Parr & Russell, 1995) and the Witness algorithm.

As we have noted above, it is possible to view eligibility traces as a type of short-term memory, since they update the latest state-action trajectory of the agent, and can therefore be used (Loch & Singh, 1998) to augment the ability to handle partial observability. Loch and Singh explored problems where a memoryless optimal policy exists and demonstrated that Sarsa(λ) (Algorithm 5) can learn an optimal policy for such domains.

Baird and Moore (Baird, 1999; Baird & Moore, 1998) show that the VAPS algorithm (see Section 3.1.4 for details) will converge for environments with partial observability where a memoryless policy exists. They did not, however, show VAPS to work better than other algorithms, either in full or partial observability.

(Hayashi & Suematsu, 1999) adapt classifier systems — rule-based systems that learn the rules online — to partial observability. They implemented a generalization of a classifier system and experimented with it, showing it to converge on a number of problems where a memoryless

policy can be calculated. They did not compare the performance of their algorithm to any other model-free technique.

3.1.6 Model-free Methods for POMDPs

Model-free methods (McCallum, 1995; Peshkin, Meuleau, & Kaelbling, 1999; Aberdeen, 2003) try to learn how to act by learning the state space, but not the other unknown parameters of the model. All such methods employ some type of internal memory to augment the observation space.

A state is therefore a combination of the most recent observation and the current internal state. Given the identifying state space, most model-free methods apply some type of model-free method from the fully observable literature, such as Q-learning or SARSA.

The differences between model-free methods are mainly with respect to defining and learning the internal state space. We use the term internal for any mechanism that the agent can modify directly. Any other part of the environment that is modified through the agents actions is called external. We review below some of the main approaches for using internal memory to solve the problem of perceptual aliasing. Figure 3.2 shows an agent augmented with an internal memory module.

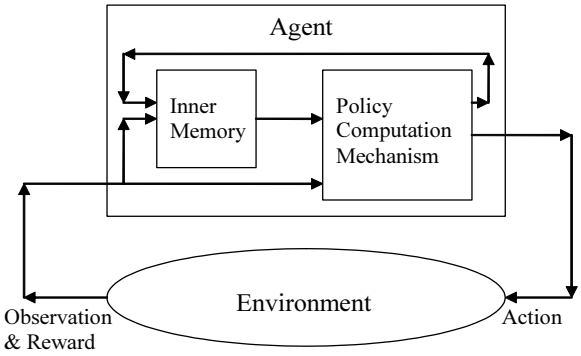


Figure 3.2: An agent model with an internal memory module.

When the agent sensors have deterministic output, disambiguating the perceptually aliased states causes the POMDP to reduce to an MDP. Since all strategies to obtain a good policy from the learned model described below originate from the MDP domain (such as Q-learning and its variations), they perform well once the POMDP was reduced to an MDP. However, when the sensors provide noisy output, these algorithms produce suboptimal policies.

Finite size histories

Researchers advocating the use of memoryless policies suggested using the last k observations instead of the immediate observation solely. The size of the history window in such cases is fixed for all observation sequences, given as a parameter rather than computed online.

For an environment with perceptually aliased states, (Littman, 1994) showed that his branch-and-bound algorithm using a history window was able to find an optimal solution, but its performance was not compared to other algorithms.

(Loch & Singh, 1998) also study domains where short-term memory policies are needed, and show that Sarsa(λ) using the last $2-3$ observations, learns faster than Littman's branch-and-

bound algorithm (Littman, Cassandra, & Kaelbling, 1995a) and achieves equal or better results than the approximation model-based algorithms in (Littman, Cassandra, & Kaelbling, 1995b) with much less computational cost. Their methods showed, though, inferior performance to stochastic policies (Williams & Singh, 1998).

Finite-state controllers

A Finite State Controller (FSC) captures finite-size history windows smartly, by learning the optimal distinctions needed to represent the best policy graph, given its size. The FSC can be learned directly without a model of the environment. (Meuleau, Peshkin, Kim, & Kaelbling, 1999b) show FSCs to outperform reactive policies computed by the VAPS algorithm, but did not compare FSCs to any other memory-based technique.

Variable length histories

The use of an arbitrarily defined history of finite size does not seem like a reasonable solution to the problem of perceptual aliasing. Agents can not be expected to know how long they need to remember their action-observation-reward trajectories. It is also usually the case that in some areas of the state space the agent needs to remember more, while in other locations a reactive policy is sufficient. A better approach would be to allow the agent to learn online the history length needed to decide on the best action in the current location.

McCallum extensively handles those issues in his dissertation (McCallum, 1996); for a detailed explanation see Section 3.1.6. His methods maintain the full set of past experiences (instances), clustering them together by various methods in order to learn to predict future rewards. He showed his algorithms to converge very fast on some problems with perceptual aliasing, and to produce superior results to the model-based perceptual distinctions approach of (Chrisman, 1992) and to the neural network method Recurrent-Q (Lin & Mitchell, 1992a, 1992b).

Bayesian learning

Suematsu et al. (Suematsu, Hayashi, & Li, 1997; Suematsu & Hayashi, 1999) pointed out some problems with McCallum's methods:

- Keeping the full set of past instances can be impractical for tasks that require a long learning curve.
- Adding noisy sensors to the task definition might cause the clustering to fail to converge (though they did not demonstrate this problem).
- The learned clusters depend on the accuracy of the statistical tests, which might cause an overfit or underfit of the data.

The Bayesian learning approach requires the agent to maintain a set of candidate models, using at each step the most suitable one to compute a policy. To avoid the problems above, Suematsu et al. suggested using Bayesian learning, where the candidate models are all the possible history trees — trees with the same structure as McCallum's USM, without the learning procedure. They presented an algorithm that smartly selects the current best fit history tree model, without enumerating all the possible trees explicitly. Their algorithm is incremental and has tractable computations at each step.

Suematsu et al. provided some experimental results to demonstrate the ability of their algorithm to solve some basic POMDP environments, but compared their algorithm only to

history trees with fixed depths that are not incrementally built. They provided no comparison between their algorithm and other memory-based algorithms.

Memory bits

Another possible approach to handle perceptual aliasing is to augment the observations with some internal memory bits (Peshkin et al., 1999). The state s of the world is therefore composed of both the current observation o and the current memory state m. The agent's action space is extended with actions that modify the current memory state by changing one of the memory bits — for every bit in memory an action is added for flipping the state of the bit on and off. The agent can apply some type of learning mechanism (such as Sarsa(λ) or VAPS) to learn the proper action, including when to change the state of the memory. The algorithm is modified so that no cost or decay is applied to the actions that modify the memory.

This approach is superior to using finite or variable history length as it can remember a meaningful event that occurred arbitrarily far in the past. Keeping all the possible trajectories from the event till the outcome is observed might cost too much, and McCallum's techniques are unable to group those trajectories together to deduce the proper action. A downside to the general memory bit approach is that failing to impose any structure on the memory (such as a Finite State Controller (Meuleau et al., 1999a)) may result in a long learning curve. Peshkin et al. 1999 demonstrated that their algorithms converge, but did not show them to be superior to any other work.

(Lanzi, 2000) also studied the effect of adding memory bits, and compared Q-learning and $Q(\lambda)$-learning — Q-learning with eligibility traces — with and without internal memory, concluding that eligibility traces and memory bits improve the performance of the agent. Lanzi did not compare his methods with other memory based techniques. Lanzi also noted that exploration of the actions that modify the memory bits should be done more often than exploration of actions that modify the world.

Long short-term memory

An appealing approach to adding an internal memory component to the agent is the use of neural networks. (Lin & Mitchell, 1992a) presented the Recurrent-Q algorithm, that uses a recurring neural network to learn to distinguish between perceptually aliased states. Long short-term memory (LSTM) is a special recurrent neural network, with some specifically designed components called memory cells that learn to capture relevant features. LSTM was compared to Lin and Mitchells former neural network approaches and to the memory bits approach of Peshkin et al. (1999) and found to converge to the optimal solution faster than both. LSTM is also the only method, to the best of our knowledge, that demonstrates model-free learning with noisy sensors. Most other techniques above assume either deterministic sensors (e.g., Peshkin et al., 1999) or sensors with very little noise (e.g., McCallum, 1996).

Hierarchical approaches

HQ-Learning (Wiering & Schmidhuber, 1997) applies to domains where the agent's task can be split into sub-tasks. In each sub-task the agent can use a reactive policy, but when transitioning from one sub-task to another, the reactive policy changes. The perceptually aliasing problem is hence solved by placing perceptually aliased states in different tasks. HQ-Learning automatically learns the proper way to split the domain and solves the sub-tasks using Q-learning with eligibility traces. Wiering and Schmidhuber showed that their algorithm outperformed the

standard learning algorithm with eligibility traces, but did not compare its performance to any other memory-based approaches.

(Hernandez & Mahadevan, 2000) implemented algorithms from hierarchical reinforcement learning (namely, the Hierarchy of Abstract Machines framework — HAM (Parr & Russell, 1997)) to introduce levels of abstraction to the solution. Their methods augment short-term memory techniques avoiding problems such as the similarity of history suffixes while moving back and forth in the same corridor. One of the building stones of their algorithm was the use of some memory technique (they used NSM — one of McCallum's earlier algorithms). The HSM algorithm they implemented was compared to NSM and showed superior results, but was not compared to either McCallum's USM or any other memory-based techniques. It is, however, likely that incorporating any other basic short-term memory technique into the hierarchical method, HSM will be able to outperform the basic method. Hierarchical approaches seem to be very promising in real world applications, but they would still probably need to use a smart short-term memory as a subroutine.

Instance-Based Learning

As some of our algorithms we present improve on McCallum's instance-based learning algorithms, we provide some deeper background into McCallum's methods. Instance-based learning algorithms store all past interactions of the agent with the environment in the form of tuples of action-reward-observation known as instances. Formally an instance is a tuple:

$$T_t = \langle T_{t-1}, a_t, r_t, o_t \rangle \tag{3.12}$$

where T_{t-1} is the previous instance and $T_0 = \perp$.

In the following sections we note some minor improvements that we added to McCallum's algorithms which were used both in our base implementation and in the various augmentations to the algorithms.

Nearest Sequence Memory

The first instance-based algorithm McCallum suggested is Nearest Sequence Memory (NSM). In NSM the agent stores Q-values for all instances, i.e. for each instance T_t and action a we store $Q(T_t, a)$. We use some similarity metric between instances to compute the k nearest neighbors of an instance. McCallum suggest using backward identity of instances:

$$sim(T_i, T_j) = \begin{cases} 1 + sim(T_{i-1}, T_{j-1}) & , \quad a_i = a_j, o_i = o_j, r_i = r_j \\ 0 & , \quad otherwise \end{cases} \tag{3.13}$$

(Nikovski, 2002) suggested the use of other similarity metrics in an offline process but these techniques are inapplicable for the online applications we focus upon.

Once the list of k neighbors is computed, each neighbor updates the Q-values of the current instance using its own Q-values. The agent then selects the action with maximal Q-value, using some exploration technique such as ϵ-greedy and executes it, observing the next reward and observation. The agent then updates the Q-values for the k neighbors using the standard Q update rule (Equation 2.4).

McCallum demonstrated NSM to converge to an optimal policy on some examples where model-based techniques based on the Baum-Welch algorithm (see Section 3.1.7 for details) took an order of magnitude more time to converge, or failed to converge at all. He also showed that NSM learns faster than the Recurrent-Q neural network approach (Lin & Mitchell, 1992b). (Estelle, 2003) also showed NSM to learn faster than Q-learning with a fixed memory window as

Algorithm 6 NSM(k)

Input: k - the number of nearest neighbors
Initialize: $t = 0$, $T_0 = \perp$
loop
  Compute τ - the list of k instances that are the nearest neighbors of T_t
  for all $T_i \in \tau$ **do**
    for all $a \in A$ **do**
      $Q(T_t, a) \leftarrow Q(T_t, a) + \frac{Q(T_i, a)}{k}$
  Choose action a using an ϵ-greedy policy on the Q-values of the current instance T_{t-1}
  Execute a, observe reward r and observation o
  for all $T_i \in \tau$ **do**
    for all $a \in A$ **do**
      Update: $Q(T_i, a) \leftarrow (1 - \alpha)Q(T_i, a) + \alpha(r + \gamma max_{a' \in A} Q(T_{i+1}, a'))$
  $t \leftarrow t + 1$
  Create new instance $T_t = \langle T_{t-1}, a, r, o \rangle$

the problem size increases. (Alexandrov, 2003) also showed NSM to properly learn to predict[1] the next perception in a real robot navigation task, outperforming bi-grams predictive model.

NSM learns a good policy very fast, but when noise is present, it fails to identify important similarities of sequences due to a noisy precept that blocks the identical experience recursive similarity. NSM also cannot cleverly identify regions of the state space where it needs to look farther into the past to find important distinctions and other areas where a simple reactive policy is sufficient, and therefore introduces unnecessary similarity calculations in such simple regions.

Utile Suffix Memory

Utile Suffix Memory creates a tree structure, based on the well known suffix trees for string operations. This tree maintains the raw experiences and identifies matching suffixes. The root of the tree is an unlabeled node, holding all available instances. Each immediate child of the root is labeled with one of the observations encountered during the test. A node holds all the instances $T_t = \langle T_{t-1}, a_{t-1}, o_t, r_t \rangle$ whose final observation o_t matches the observation in the node's label. At the next level, instances are split based on the last action of the instance a_t, then on (the next to last) observation o_{t-1}, and so forth. All nodes act as buckets, grouping together instances that have matching history suffixes of a certain length. Leaves take the role of states, holding Q-values and updating them. The deeper a leaf is in the tree, the more history the instances in this leaf share.

The tree is built online during the test run. To add a new instance to the tree, we examine its precept and follow the path to the child node labeled by that precept. We then look at the action prior to this precept and move to the node labeled by that action, and then branch on the precept prior to that action and so forth, until a leaf is reached. For example, the maze in Figure 3.1(a) might generate the tree in Figure 3.3.

Identifying the proper depth for a certain leaf is a major issue. Leaves should be split if their descendants show a statistical difference in expected future discounted reward associated with the same action. Instances in a node should be split if knowing where the agent came from helps predict future discounted rewards. Thus, the tree must keep what McCallum calls fringes, i.e., subtrees below the "official" leaves.

[1] Alexandrov did not use NSM for control, only for prediction, therefore not keeping track of Q-values.

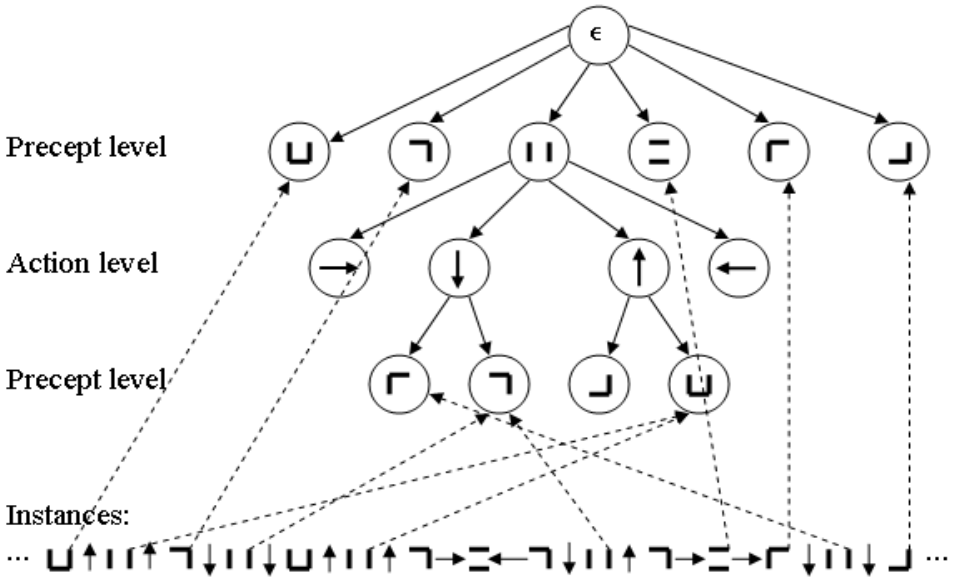


Figure 3.3: A possible USM suffix tree generated by the maze in Figure 3.1(a). Below is a sequence of instances demonstrating how some of the instances are clustered into the tree leaves.

For better performance, McCallum did not compare the nodes in the fringes to their siblings, only to their ancestor "official" leaf. He also did not compare values from all actions executed from the fringe, only the action that has the highest Q-value in the leaf (the policy action of that leaf). To compare the populations of expected discounted future rewards from the two nodes (the fringe and the "official" leaf), he used the Kolmogorov-Smirnov (KS) statistical test — a non-parametric test used to find whether two populations were generated by the same distribution. If the test reported that a statistical difference was found between the expected discounted future reward after executing the policy action, the leaf was split, the fringe node would become the new leaf, and the tree would be expanded to create deeper fringes.

Instead of comparing the fringe node to its ancestor "official" leaf, we found it computationally possible to compare the siblings of the fringe, avoiding the problem that the same instance appears in both distributions. McCallum compared only the expected discounted future rewards resulting from executing the policy action, where we compare all the values following all actions executed after any of the instances in the fringe. McCallum used the KS test, where we choose to use the more robust randomization test (Yeh, 2000) that works well with small sets of instances. McCallum also considered only fringe nodes of certain depth, given as a parameter to the algorithm, where we choose to create fringe nodes as deep as possible, until the number of instances in the node diminished below some threshold (we used a value of 10 in our experiments).

The expected future discounted reward of instance T_i is defined by:

$$Q(T_i) = r_i + \gamma U(L(T_{i+1})) \tag{3.14}$$

where $L(T_i)$ is the leaf associated with instance T_i and $U(s) = max_a(Q(s,a))$.

After inserting new instances to the tree, we update Q-values in the leaves using:

$$R(s,a) = \frac{\sum_{T_i \in T(s,a)} r_{i+1}}{|T(s,a)|} \tag{3.15}$$

$$Pr(s'|s,a) = \frac{|\forall T_i \in T(s,a), L(T_{i+1}) = s'|}{|T(s,a)|} \tag{3.16}$$

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} Pr(s'|s,a)U(s') \tag{3.17}$$

This corresponds to a single step of the value iteration algorithm used in MDPs.

Now that the Q-values have been updated, the agent chooses the next action to perform based on the Q-values in the leaf corresponding to the current instance T_t:

$$a_{t+1} = argmax_a Q(L(T_t), a) \tag{3.18}$$

McCallum used an ϵ-greedy exploration strategy; with a probability of $1 - \epsilon$ the best action is executed and with a probability of ϵ a random action is selected. In our implementation we used the same strategy.

3.1.7 Model-based Methods for POMDPs

When the model of the environment is unknown, it is possible to learn the model and then solve it using one of the solution techniques in Section 2.2.2. Model-based methods use past experience in the form of a sequence of instances — $\langle a_t, o_t, r_t \rangle$ — to learn a POMDP model that is likely to generate the sequence. Researchers have not favored this approach since solving POMDPs has always been a bottleneck that needed addressing, and adding to that the difficulties of learning the model seemed to make the problem infeasible. Modern techniques, however, allow for a tractable solution for POMDPs with reasonably sized state spaces, making future research in the area of model-based methods more attractive.

The Baum-Welch Algorithm

Hidden Markov models are used to model dynamic systems where the transition to the next state may depend on a hidden (unobserved) variable. A hidden Markov model can be viewed as a POMDP without actions. It has been noted (Chrisman, 1992; McCallum, 1993; Nikovski & Nourbakhsh, 2000) that the Baum-Welch algorithm (e.g., Bilmes, 1997) for learning hidden Markov models can be applied to POMDPs. We can fix the structure of the POMDP (the number of states) and learn the state transition probabilities, given the observations. The traditional approach then uses the EM algorithm to recalculate a new state space that maximizes the log-likelihood of the observations and then the Baum-Welch algorithm is executed again (e.g., Shatkay, 1999).

Chrisman implemented a component containing a POMDP model that predicts the current state, and updated it online using the Baum-Welch algorithm. A variation of Q-learning was used to compute Q-values given the states predicted by the POMDP predictive model, rather than solving the model. States of the POMDP could be split if the need for finer distinctions was detected. Chrisman experimented with noisy sensors and actions with stochastic results but did not compare his algorithm to any other learning method.

Maintaining a model by splitting the states that required refining in order to perform better was further explored by (McCallum, 1993, 1996) in his Utile Distinctions Memory (UDM) algorithm. McCallum initialized a model that has one state for each observation. States were then split using a statistical test on the future discounted reward of instances, given the prior

state. A state is split if the statistical test showed that knowing where (which state) the agent came from helped predict future reward. The transition probabilities and policy were calculated in the same manner suggested by Chrisman. Like Chrisman, McCallum showed his algorithm to converge on a domain with some noise, but did not compare it to any other algorithm.

(Wierstra & Wiering, 2004) recently suggested augmenting McCallum's UDM by using an improved policy calculation algorithm that uses belief space to update Q-values and then applies a form of the Q_{MDP} approximation technique to calculate the next action. They split states when a statistically significant difference in discounted future rewards is detected using the EM algorithm to split instances into mixture components. Their algorithm is split into an offline learning stage when state splitting is executed and model parameters are re-estimated, and an online acting stage where the belief state is maintained and Q-values updated. Their algorithm showed inferior performance to UDM on a small maze world, but was able to solve a medium sized maze, with performance similar to the RL-LSTM recurrent network algorithm with sensor accuracy of 0.7.

All algorithms that use the Baum-Welch algorithm cannot escape its time-heavy complexity $O(N^2)$ on each iteration. Using Baum-Welch to estimate the model parameters, therefore, seems an impractical choice for large domains. A second problem with EM and Baum-Welch is their inability to avoid local minima.

State merging methods

The algorithms we reviewed in the previous section were initialized with a small number of states that were split if the need arose. Nikovski et al. (Nikovski & Nourbakhsh, 2000; Nikovski, 2002) suggested starting with a state for each observed instance and merging them until a sufficient number of states has been reached. An initial state, corresponding to instance T_i has a probability of 1 of moving to the state representing instance T_{i+1} with a single action, single reward, and single observation, but as states are merged the transition, reward, and observation functions are enriched.

The first criterion Nikovski examined for merging states was the minimal decrease in log-likelihood. The initial model maximizes the log-likelihood, and every merge causes some decrease in this measure. Nikovski tried greedy merging of pairs of states that caused the minimal decrease.

Nikovski then applied one of McCallum's earlier model-free methods for discovering similar history suffixes — the Nearest Sequence Memory (NSM) algorithm — to cluster together instances. NSM identifies for every instance k other instances that best agree on the suffix of their history and uses their Q-values to update the current instance Q-values. Nikovski clustered together the states corresponding to these instances, using some enhancements such as also measuring how much the instances agree in their future, not just their history.

Nikovski implemented these approaches for learning POMDPs and compared them to implementations of the Baum-Welch and Steepest Gradient Ascent in log-likelihood (Binder, Koller, Russell, & Kanazawa, 1997) — algorithms where the number of states is predefined and the agent needs to learn the transition, reward, and observation function. The resulting POMDPs were solved using approximation techniques such as Q_{MDP}. Nikovski did not report any attempt to apply any technique for fully solving the POMDP to the learned models. Experiments were conducted on a robot simulator with very little noise, but with perceptual aliasing. The results of the experiment show the enhanced NSM-based merging technique to produce slightly superior results to the Baum-Welch and Steepest Gradient Ascent methods. Nikovski did not compare his methods to any model-free algorithm.

3.2 Model-Free Learning Under Noisy Sensors[2]

As mentioned previously, when the environment suffers from perceptual aliasing, yet the agent's sensors are deterministic, resolving the perceptual aliasing using one of the methods described in Section 3.1.5 reduces the problem to a Markov Decision Process, whose states are not necessarily the states of the MDP underlying the POMDP.

For example, we can use the leaves of the tree created by the USM algorithm as the states of the MDP. In the maze in Figure 3.1(a), USM may create two states (leaves) corresponding to the observation of walls east and west of the agent (| |), the first preceded by the action "down" and beforehand observing a wall above and to the left, and the second preceded by an "up" action following observing a wall below and to the right (see Figure 3.3). These two states correspond to the same state (the left corridor) in the "trivial" MDP formulation of the maze. Another such pair of states will correspond to the right corridor in the maze. On the other hand, in the maze in Figure 3.1(b), the top-leftmost and top-rightmost cells will correspond to the same state (leaf) in the USM tree.

The resulting MDP is therefore different from the MDP a human might construct using a map of the maze, yet the model is valid and can be solved by any MDP policy computation technique such as Q-learning. Environments with perceptual aliasing but without noisy sensors can be regarded as a restrictive class of the POMDP framework. In general, however, resolving the perceptual aliasing would still be insufficient for the computation of an optimal solution. Ignoring the sensor noise will result in a solution that is optimal for a noiseless environment yet its performance in the "real" environment is unknown.

The classical definition of reinforcement learning under partial observability assumes that the agent only knows about the actions it can perform prior to the learning process. However, it is reasonable to assume that the agent also has predefined knowledge over its sensor model. As we consider the sensors to be a part of the agent's hardware, the accuracy of the sensors can also be known a priori without any learning.

In this section we explore methods that use a predefined sensor model to improve some well-known techniques for RL in partially observable domains.

3.2.1 Performance of Model-Free Algorithms in Noisy Domains

We evaluate standard model-free techniques in partially observable domains. These techniques first disambiguate the perceptual aliasing, and then compute an optimal policy for the resulting fully observable MDP. We begin by showing empirically how the performance of such techniques degrades sharply as noise in the sensors increases.

Two maze worlds (seen in Figure 3.1(a) and 3.1(b)), identical to the worlds upon which McCallum demonstrated the performance of the USM algorithm, were used to test the model-free algorithms. In both cases, some of the world states are perceptually aliased and the algorithm should learn to identify the "real" world states. The agent in our experiments has four independent sensors allowing it to sense an immediate wall above, below, to the left, and to the right of its current location. Sensors output a boolean value with probability α of being correct. Thus, the probability of all sensors providing the correct output is α^4. In both maze domains there is a single location that grants the agent a reward of 10 (a goal state). Upon receiving that reward, the agent is transformed to any other state in the maze randomly. If the agent bumps into a wall it pays a cost (a negative reward) of 1. For every move the agent pays a cost of 0.1.

We compare the performance of applying finite size history windows to Q-learning and Sarsa, eligibility traces, memory bits, and USM on the two worlds. In Tables 3.1 and 3.2, QL_2

[2]Parts of this section appeared in (Shani & Brafman, 2004)

denotes using Q-learning with a history window of size 2, and S_2 denotes using Sarsa with a window size of 2. $S(\lambda)$ denotes the Sarsa(λ) algorithm. Adding the superfix 1 denotes adding 1 bit of internal memory. For example, $S(\lambda)_2^1$ denotes using Sarsa(λ) with a history window of 2 and 1 internal memory bit. The columns and rows marked σ^2 present the average variance over methods (bottom row) and α values (right-most column). Tables 3.1 and 3.2 also include results for the NUSM algorithm defined in the next section.

As we are interested only in the effect of noisy sensors, the maze examples we use do not demonstrate the advantages of the various algorithms; USM's ability to automatically compute the needed trajectory length in different locations and the internal memory ability to remember events that occurred arbitrarily far in the past are unneeded since our examples require the agent to look only at the last 2 instances in every perceptually aliased location.

We ran each algorithm for 50000 steps, learning the model as explained above and calculated the average discounted reward. We ran experiments with varying values of α (accuracy of the sensors) ranging from 1.00 (sensor output is without noise) to 0.85 (overall output accuracy of 0.522). Reported results are averaged over 10 different executions of each algorithm. We also ran experiments for Sarsa and Q-learning with only the immediate observation, which yielded poor results (as expected), and for history windows of size 3 and 4 which resulted in lower performance than a history window of size 2 for all algorithms (and are therefore unreported in the tables). Additional memory bits did not improve performance either.

As Tables 3.1 and 3.2 clearly show, the performance of all algorithms degrades sharply as sensor noise increases. When individual sensor accuracy is 0.85 — meaning that 85% of the observations provided by the sensor are accurate — the performance of most methods drops to under 1/3 of their initial performance.

3.2.2 Incorporating a Sensor Model

In the coming section we diverge slightly from the classical definition of the reinforcement learning problem. A robot operating in an environment has access not only to its motors but also to its sensors. It is therefore realistic to assume that the robot has some prior knowledge of the sensor accuracy. A touch sensor, for example, can be tested offline to compute the probability that it detects a wall when the robot hits it. Such experiments can be conducted in any controlled environment and do not require the robot to operate in the environment it should learn.

We hence assume that the reinforcement learning agent is augmented with a *sensor model* that provides $pr(o' = o)$ — the probability that observation o' was generated instead of the 'true' observation o due to noisy sensors.

For example, the agent has n boolean sensors with an accuracy probability p_i for sensor i. A single observation is composed of n output values $o = \langle \omega_1, ..., \omega_n \rangle$, where $\omega_i \in \{0, 1\}$. Once two observations o_1 and o_2 were detected, where $\omega_{1i} = \omega_{2i}$ for some sensor i designed to detect the state of a certain feature of the environment, there can be two possible cases where the two observations were generated by the same feature:

- Both observations were successful, yielding a probability of p_i^2.
- Both observations failed, with a probability of $(1 - p_i)^2$.

hence the probability that the two observations came from the same feature state is $p_i^2 + (1-p_i)^2$.

When $\omega_{1i} \neq \omega_{2i}$, the two observations may still correspond to the same feature state if one of them was faulty (with a probability of $1 - p_i$) and the other was true (with a probability of p_i). As any one of the sensors might be the faulty one, the probability of the two observations coming from the same feature state is $2p_i(1 - p_i)$.

Assuming that sensors are independent, we can now define the similarity between two observation vectors $o_1 = \langle \omega_{11}, ..., \omega_{1n} \rangle$ and $o_2 = \langle \omega_{21}, ..., \omega_{2n} \rangle$:

$$sim(o_1, o_2) = \prod_{i=1}^{n} sim_i(\omega_{1i}, \omega_{2i}) \tag{3.19}$$

$$sim_i(\omega_{1i}, \omega_{2i}) = \begin{cases} p_i^2 + (1-p_i)^2 & , \quad \omega_{1i} = \omega_{2i} \\ 2p_i(1-p_i) & , \quad \omega_{1i} \neq \omega_{2i} \end{cases} \tag{3.20}$$

Given such a sensor model, we can augment the basic instance-based learning techniques such as NSM and USM to learn from *similar* instances, where the original algorithms could use only *identical* instances. We do so by modifying the deterministic instances similarity metric (Equation 3.13):

$$sim_{stochastic}(T_i, T_j) = simlength(T_i, T_j) \cdot pr(T_i = T_j) \tag{3.21}$$

$$pr(T_i = T_j) = \begin{cases} pr(o_i = o_j) \cdot pr(T_{i-1} = T_{j-1}) & , \quad a_i = a_j, r_i = r_j \\ 1 & , \quad otherwise \end{cases} \tag{3.22}$$

$$simlength(T_i, T_j) = \begin{cases} 1 + simlength(T_{i-1} = T_{j-1}) & , \quad a_i = a_j, r_i = r_j \\ 0 & , \quad otherwise \end{cases} \tag{3.23}$$

when $pr(o' = o)$ is defined by the sensor model. This metric defines the expectation of the equality of the instances' suffixes. The above equation can be expanded to the formal POMDP sensor model where actions depend both on the latest state and the latest action by replacing $pr(o = o')$ with $pr(o = o'|a)$.

The agent might also possess similar knowledge of an action model that defines "action similarity". That is, the agent might know *a priori* that when it tries to go up it sometimes goes left (e.g., due to slippery floor) and may be able to measure the prior probability of that event offline. Given such a model for stochastic actions or stochastic rewards, Equation 3.21 can be easily extended to express such information.

3.3 Utile Suffix Memory in the Presence of Noise[3]

There are two types of noise in perception:

- *False-negative* — The agent might receive different observations at the same location. For example the agent might visit the top-left cell in the maze in Figure 3.1(a) twice, once receiving the (true) observation of a wall above and to the left, and once observing just a wall to the left, since the sensors failed to identify the wall above.
- *False-positive* — The sensors might produce identical observations at different locations. For example, the agent might only observe a wall above in all top three cells in the maze since the sensors failed to identify the second wall in each cell.

USM handles false positives by differentiating identical current observations using the agent's history. Knowing where the agent came from, may help it differentiate between two possible locations, even though the two provide the same observations.

However, USM does not handle false negative perceptions well. When the agent is at the same state, yet it receives different observations, the agent is unable to learn from the noisy instance. The agent cannot use any data in sequences that contain noisy observations to learn

[3]Parts of this section appeared in (Shani & Brafman, 2004)

about the "real" state and hence much of the available information is lost. Our Noisy Utile Suffix Memory (NUSM) is designed to overcome this weakness.

USM inserts each instance into a single path, until a leaf is reached. When sensors have stochastic noisy output, we can use the agent knowledge over its sensor model to define the similarity between two observations. Instead of adding a new instance $T_t = \langle a_{t-1}, o_t, r_t \rangle$ to a single path in the tree, we insert it into several paths with different weights. We define $w_s(T_t)$ to be the weight of instance T_t in node s. We begin with $w_{root}(T_t) = 1$. We use the following method for inserting an instance T_t with weight w into a node at depth k:

- **action node** — we insert the instance into every child node c (labeled by an observation), with weight $w \cdot sim(o_{t-k-1}, label(c))$.
- **observation node** — we insert the instance only into the child c labeled by action a_{t-k-1} with the same weight w (assuming we do not have data over the actions accuracy model).

We denote $w_s(T_t)$ as the weight of instance T_t in node s. The weights of the instances are stored in each node together with the instance.

We can now rewrite Equations 3.15 and 3.16 as:

$$R(s,a) = \frac{\sum_{T_i \in T(s,a)} r_{i+1} \cdot w_s(T_i)}{\sum_{T_i \in T(s,a)} w_s(T_i)} \tag{3.24}$$

$$Pr(s'|s,a) = \frac{\sum_{T_i, L(T_{i+1})=s'} w_s(T_i)}{\sum_{T_i \in T(s,a)} w_s(T_i)} \tag{3.25}$$

As data in the leaves is now weighted, the randomization test we use to decide whether a leaf should be split should be calculated using a statistical mean of the expected future discounted rewards.

Storing un-identical, but similar, sequences in a state allows it to use data containing false negative noisy observations to supply information about the "real" sequence. When an instance that originates in a different area of the maze, yet noise causes it to initially appear relevant, is inserted deeper into the tree, its weight will diminish considerably and will be dominated by the weights of the relevant sequences. The sequences that do not reflect false negative observations but rather other areas of the state space are hence filtered until the weight of the relevant instances becomes dominant.

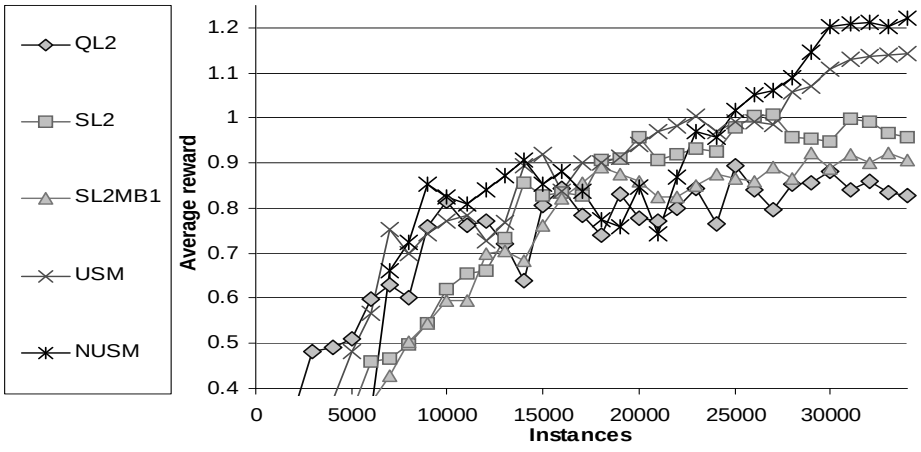


Figure 3.4: Convergence rates for the maze in Figure 3.1(a) when sensor accuracy is 0.9.

α	QL_2	S_2	$S(\lambda)_1$	$S(\lambda)_2$	$S(\lambda)_3$	$S(\lambda)_1^1$	$S(\lambda)_2^1$	USM	NUSM	σ^2
1.00	1.51	1.54	0.32	1.53	0.98	1.27	1.53	1.56	**1.57**(+0%)	0.033
0.99	1.46	1.47	0.33	1.47	0.98	1.34	1.45	1.49	**1.54**(+3%)	0.016
0.98	1.42	1.42	0.32	1.36	0.83	1.41	1.40	1.42	**1.44**(+1%)	0.008
0.97	1.36	1.38	0.41	1.31	0.64	1.35	1.35	1.38	**1.43**(+4%)	0.007
0.96	1.28	1.26	0.38	1.29	0.58	1.24	1.27	1.35	**1.40**(+4%)	0.014
0.95	1.24	1.23	0.43	1.21	0.55	1.26	1.22	1.30	**1.35**(+4%)	0.009
0.94	1.11	1.10	0.47	1.16	0.45	0.89	1.12	1.18	**1.29**(+9%)	0.025
0.93	1.05	1.03	0.42	1.13	0.43	0.94	1.07	1.16	**1.29**(+11%)	0.023
0.92	0.96	0.88	0.47	1.10	0.33	0.92	1.04	1.12	**1.20**(+7%)	0.014
0.91	0.88	0.82	0.47	1.06	0.29	0.74	0.94	0.96	**1.12**(+17%)	0.022
0.90	0.83	0.73	0.46	1.02	0.22	0.77	0.92	0.99	**1.07**(+8%)	0.013
0.89	0.74	0.60	0.48	0.95	0.23	0.80	0.87	0.93	**1.04**(+12%)	0.015
0.88	0.61	0.59	0.42	0.90	0.14	0.66	0.83	0.84	**1.01**(+20%)	0.013
0.87	0.58	0.50	0.48	0.85	0.12	0.63	0.78	0.71	**0.98**(+38%)	0.011
0.86	0.40	0.37	0.46	0.79	0.07	0.55	0.76	0.57	**0.92**(+61%)	0.021
0.85	0.45	0.35	0.45	0.75	0.05	0.47	0.68	0.47	**0.87**(+85%)	0.018
σ^2	0.01	0.015	0.004	0.004	0.022	0.061	0.005	0.02	0.003	

Table 3.1: Average discounted reward as a function of sensor accuracy, for the maze in Figure 3.1(a).

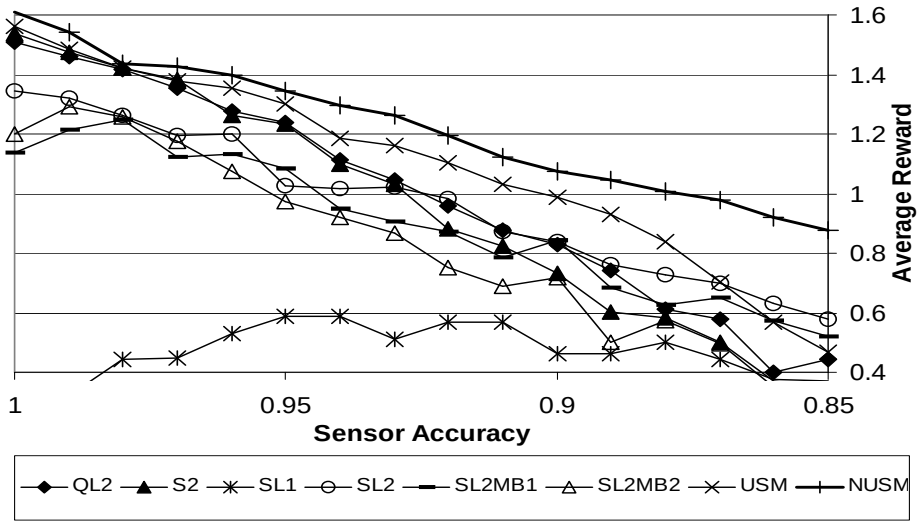
α	QL_2	S_2	$S(\lambda)_1$	$S(\lambda)_2$	$S(\lambda)_3$	$S(\lambda)_1^1$	$S(\lambda)_2^1$	USM	NUSM	σ^2
1.00	1.42	1.46	0.23	1.53	1.49	1.47	1.54	1.75	**1.72**(-2%)	0.004
0.99	1.40	1.41	0.24	1.44	1.34	1.24	1.43	1.57	**1.61**(+3%)	0.027
0.98	1.33	1.35	0.25	1.35	1.24	0.94	1.34	1.43	**1.46**(+2%)	0.034
0.97	1.30	1.29	0.26	1.22	1.15	0.90	1.21	1.40	**1.40**(0%)	0.032
0.96	1.26	1.25	0.24	1.06	1.06	0.43	1.12	1.28	**1.31**(+2%)	0.015
0.95	1.19	1.16	0.21	1.00	0.90	0.33	1.03	1.23	**1.26**(+2%)	0.015
0.94	1.09	1.05	0.14	0.93	0.85	0.30	0.93	1.09	**1.14**(+5%)	0.011
0.93	1.05	0.94	0.12	0.82	0.76	0.39	0.77	1.09	**1.09**(+0%)	0.018
0.92	0.94	0.84	0.12	0.72	0.64	0.30	0.66	1.02	**1.03**(+1%)	0.011
0.91	0.85	0.80	0.09	0.77	0.47	0.24	0.48	0.93	**0.96**(+3%)	0.013
0.90	0.69	0.72	0.10	0.65	0.42	0.23	0.48	0.87	**0.91**(+5%)	0.013
0.89	0.64	0.58	0.06	0.58	0.24	0.24	0.40	0.81	**0.92**(+14%)	0.010
0.87	0.44	0.42	0.06	0.47	0.16	0.16	0.31	0.72	**0.82**(+14%)	0.012
0.86	0.30	0.21	0.04	0.26	0.09	0.13	0.20	0.68	**0.81**(+19%)	0.015
0.85	0.08	0.10	0.01	0.29	0.09	0.18	0.16	0.61	**0.75**(+23%)	0.008
σ^2	0.008	0.01	0.003	0.009	0.006	0.071	0.016	0.011	0.006	

Table 3.2: Average discounted reward as a function of sensor accuracy, for the maze in Figure 3.1(b).

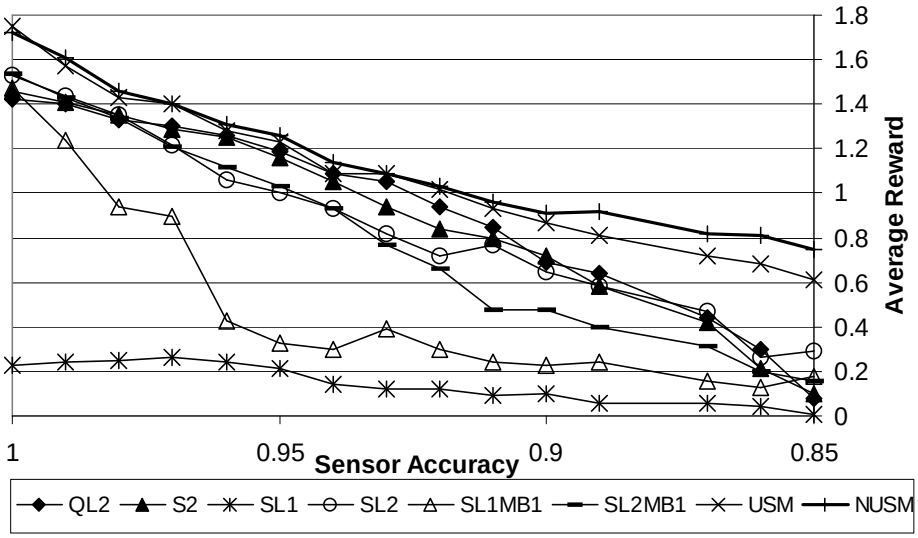
NUSM also learns a superior policy for states that correspond to a noisy sequence. Leaves that correspond to such instances can still use data from real sequences to learn the value of actions.

Figures 3.5(a) and 3.5(b) compare the performance of NUSM to USM and Q-learning with a fixed history window of length 2. When sensor accuracy is high, the performance of all algorithms is similar, but when sensor accuracy degrades, NUSM shows superior results to USM and Q-learning.

NUSM is more expensive computationally than USM and takes longer to converge. In Figure 3.4, we can see that it still converges reasonably fast. Moreover, each NUSM iteration takes about 5.6 milliseconds, when a USM iteration takes 3.1 milliseconds with the same accuracy of $\alpha = 0.85$ and a similar number of nodes ($10,229$ for NUSM and $10,349$ for USM — including fringe nodes), making NUSM reasonable for online learning.



(a) NUSM performance on the maze in Figure 3.1(a).



(b) NUSM performance on the maze in Figure 3.1(b).

Figure 3.5: The performance of NUSM when sensor accuracy degrades compared to other algorithms.

3.3.1 Discussion

USM is computing policies in the space of truncated histories. As such, we can think of USM as imposing a similarity measurement over histories, joining together histories that have identical suffixes. In that retrospect, NUSM uses an enhanced similarity measurement, joining together histories that do not share identical suffixes.

We can think of USM as trying to group together histories that correspond to the same belief state in each leaf. Then, the value computation of USM can be observed as Q-learning in the belief space MDP. However, there is no guarantee that two histories that share a finite suffix

map to the same belief state. Nevertheless, in a discounted setting, it is possible that USM policy will approach an optimal expected discounted reward given infinite amount of experience, even though leaves will continue to split and the policy would not converge.

If we view USM as attempting to group together histories that map to the same belief state, we can think of NUSM as grouping together belief states that are sufficiently similar such that their optimal action is identical. Since we do not currently have a proof that USM converges in the case of an infinite horizon, it is unlikely that we can prove NUSM to converge.

We leave a deeper discussion of the relationship between USM and NUSM and their convergence guarantees to future research.

3.4 Model-Based Offline Learning[4]

As we showed in the previous section, resolving the perceptual aliasing when the agent sensors are deterministic results in a fully observable Markov Decision Process (MDP). A solution to this model (whether by model-free or model-based techniques) can produce optimal results. When the sensors provide stochastic noisy output, solving an MDP model of the environment is merely an approximate solution, and not a very good one.

Even though we can use information from the sensor model to augment model-free methods to learn from noisy sequences, we can still expect added value when using a POMDP model of the environment. The solution for such a model is optimal where model-free methods are no more than approximations.

Learning a POMDP model can also have other benefits. Using a model of the partially observable environment can help the agent to analyze the environment. The agent can, for example, use the model to direct exploration towards promising, or less explored, regions of the world.

A model-free method in a fully observable MDP does not need to learn any environment parameters, and computes a value function or a policy directly. However, most model-free methods in the partially observable case must compute some type of state space, and only avoid the computation of the transition and observation functions for the model.

We thus suggest here leveraging the computed state space of any model-free method, and learn additionally the transition and observation functions resulting in a complete POMDP model. We can then compute a policy for this model using any technique such as a point-based algorithm (see Section 4.1). The resulting policy should have superior performance to the original, model-free policy.

Our method therefore contains the following steps:

- First, we execute a model-free method until it learns a state space.
- We then either use the already gathered experience or execute the policy of the model-free method with some exploration factor to compute the transition and observation functions.
- We use the resulting POMDP model to compute a policy using POMDP solution techniques.

In the following sections we demonstrate our approach over two model-free methods — USM and Memory Bits — and show that the policy of the resulting POMDP model has superior performance to the original model-free method as sensor noise increases. Due to the specific features of every model-free technique, the resulting POMDP is not an exact representation of the "true" environment, yet even such inaccurate POMDPs show superior performance to the original model-free policy.

[4]Parts of this section appeared in (?)

3.4.1 Creating a POMDP from the Utile Suffix Memory

Once USM has converged to a locally optimal tree structure, we can use the resulting tree structure to create a POMDP.

We define the set of leaves USM calculated to be the state space of the POMDP. A known problem for USM is that many leaves may correspond to the same environment state (see discussion in Section 3.2) and, thus, the resulting state space is not compact. We tried several techniques to merge these leaves, such as merging leaves that have similar expected rewards and merging leaves that have similar transition probabilities, but both methods either did not merge any leaves or resulted in the wrong leaves getting merged and thus in suboptimal performance.

Some of the leaves correspond to sequences that were rarely observed. These sequences are likely the result of a noisy sensor output. We therefore set a threshold to remove leaves that correspond to instances that were seldom observed. Removing a state results in gaps in the L function that maps the states to the instances in the observed sequence, i.e., some instances do not have states mapped to them. Our transition function needs to be calibrated. We therefore split those instances between all states with the same latest observation (the first observation on the path to the leaf) by the weight of these states.

Obtaining the POMDP parameters from the USM tree structure is straightforward. The actions (A) and observations (Ω) are known to the agent prior to learning the model, as we assume that the agent has a sensor model. We use the leaves of the tree as the states in S. The transition function ($tr(s, a, s')$) is defined by Equation 3.16 and the reward function ($R(s, a)$) by Equation 3.15.

To define the observation function, we use the sensor model as defined in Section 3.2.2. As any leaf s in the USM tree corresponds to a single observation $o(s)$ (the observation of the node closest to the root), we define $O(a, s, o) = pr(o = o(s))$. Such a formalization cannot capture the dependency on actions, yet it still produces a valid POMDP.

Alternatively, we can learn an observation function of the form $pr(o|a_t, s_{t-1})$ — the probability observation o is seen after executing action a_t in state s_{t-1}, instead of the traditional $O(a_t, s_t, o_t)$ function. This is useful when we have pure sensing actions, but in many cases such a POMDP definition is unnatural.

3.4.2 Creating a POMDP from Memory Bits

After the convergence of the memory bits algorithm, we can use the learned Q-table and the observed instances to initialize the POMDP state space. In the memory bits approach, observation o originates in a perceptually aliased state, if the agent learned that it needs to behave differently when seeing o based upon the internal memory state.

Formally, let us define $s = \langle o, m \rangle$ — the agent state composed of the sensor observation o and the agent internal memory state m. An observation o originates in a perceptually aliased world state if there exist two agent states $s =< o, m >$, $s' =< o, m' >$, such that $m \neq m'$ and $max_a Q(s, a) \neq max_a Q(s', a)$, and none of these actions is an action that flips a memory bit. We can merge every pair of states $s =< o, m >$ and $s' =< o, m' >$ that are not perceptually aliased, ignoring their diferences.

Using the instances we have observed, we can compute the transition $tr(s, a, s')$, reward $R(s, a)$, and observation $O(s, a, o)$ functions much the same way as we did for the USM-based model. The instances we use to learn the model must be generated by the memory bits algorithm when it is near to convergence, in order to properly learn the probabilities. When collecting these instances, the agent must still explore, but it must not use exploration in states where the optimal action modifies the memory bit, since this will cause it to observe transitions that are impossible when following the learned policy. The learned model is therefore imperfect and

does not have all transition and reward data, yet it still outperforms the original memory bits algorithm.

3.4.3 Experimental Results

In our experiments we ran the USM-based POMDP on the mazes in Figure 3.6. While these environments are uncomplicated compared to real world problems, they demonstrate important problem features such as multiple perceptual aliasing (Figure 3.6(b)) and the need for an information gain action (Figure 3.6(c)).

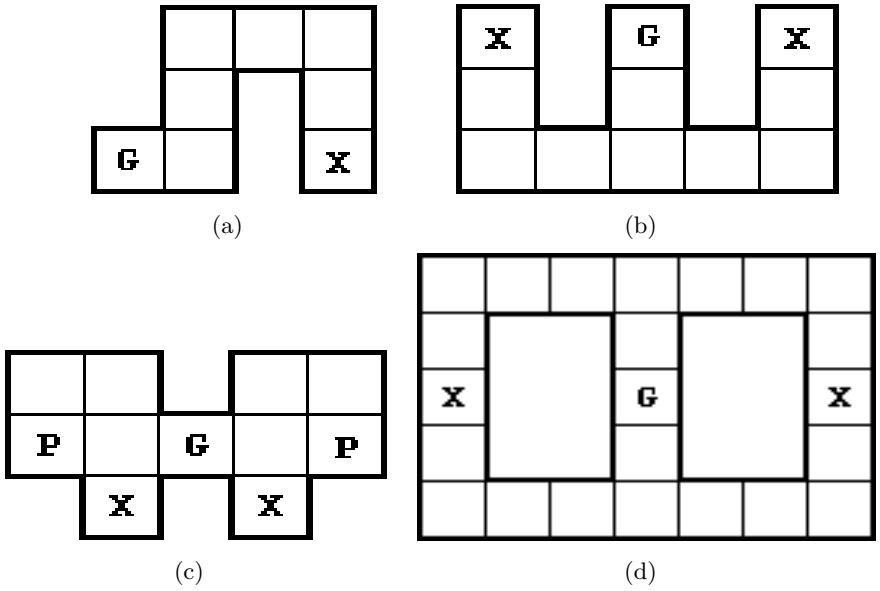
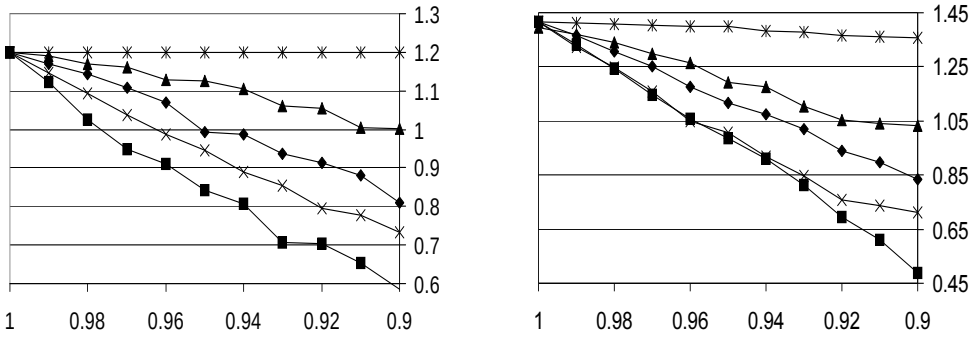


Figure 3.6: Four maze domains. The agent receives a reward of 9 upon reaching the goal state (marked G). Immediately afterwards (in the same transition), the agent is transferred to one of the X states. Arrival at a P state results in a penalty (negative reward) of 9.

We ran both USM and SARSA(λ) with one additional memory bit on the mazes in Figure 3.6. Once the average discounted reward collected by the algorithms passed a certain threshold, their current state (USM's tree structure, and SARSA's Q-table) was kept and exploration was stopped (as the POMDP policy does not explore). In all tests, the convergence of USM was much faster (about four times faster) than the corresponding memory-bits method. Then, the runs were continued for 5000 iterations to calculate the average discounted reward gained by the converged algorithm. The agent current state was then used to learn a POMDP model as explained above. The model was solved by the Perseus algorithm (Spaan & Vlassis, 2005),[5] and the resulting model was executed for another 5000 iterations. This process was repeated 10 times for each point in our graphs, and the presented results are an average of these executions. To show the optimal possible policy, we manually defined a POMDP model for each of the mazes above, solved it using Perseus and ran the resulting policy for 5000 iterations. This was done only once, as there is no learning process involved.

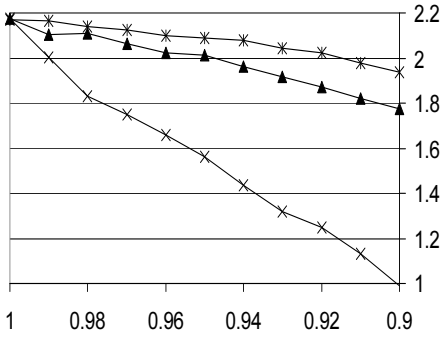
The average collected reward for each method when α (the sensor accuracy) varies from 1.0 (deterministic sensor output) to 0.9 (probability 0.65 for detecting all features correctly) is shown in Figure 3.7. SLMB1 stands for adding a single memory bit to the Sarsa(λ) algorithm. SLMB1 Model and USM Model are the output of executing the memory bits and the USM

[5]We further discuss the Perseus algorithm, as well as other point-based algorithms, in the next chapter

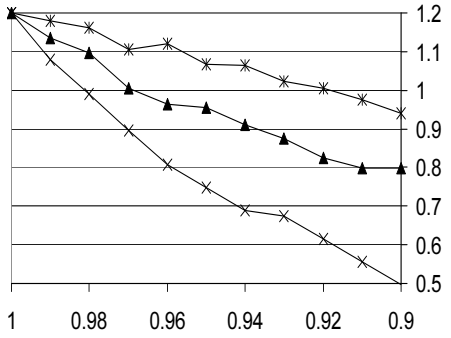


(a) Results for Figure 3.6(a)

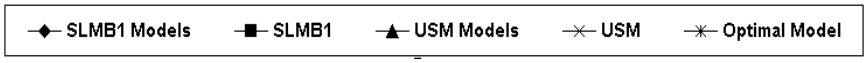
(b) Results for Figure 3.6(b)



(c) Results for Figure 3.6(c)



(d) Results for Figure 3.6(d)



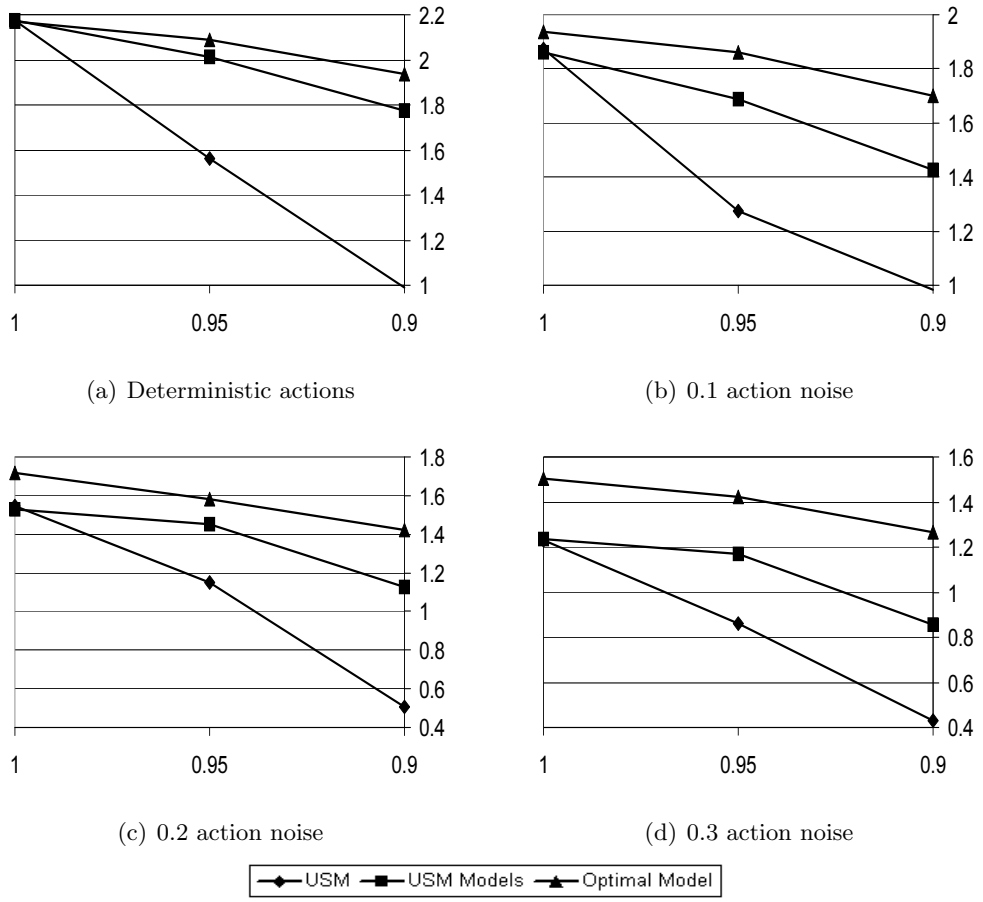
(e) Legend for the above graphs

Figure 3.7: Results for the mazes in Figure 3.6. In all the above graphs, the X-axis contains the diminishing sensor accuracy and the Y-axis marks average discounted reward per agent action. The above results are averaged over 10 different executions for each observation accuracy and method. All variances were below 0.015 and in most cases below 0.005.

algorithms, respectively, creating a POMDP from the algorithm output, solving it and executing the resulting policy. Optimal Model is the manually defined POMDP model. In the two lower mazes, the memory bits algorithm needed two memory bits and failed to converge as sensor noise increased. We thus report results only for USM and the manually defined POMDP on these domains.

As seen from the results, the model-based methods greatly improve the original model-based techniques and the advantage becomes more important as noise in the sensors increases. The memory bits-based model does not perform as well as the USM-based model, probably due to the inaccurate model parameters that were learned because of the inability to explore all states and actions. The USM model in our experiments is also suboptimal, mainly because several leaves correspond to the same world state. The number of states discovered by both algorithms are compared in Figure 3.9.

In most cases, actions in MDPs and POMDPs do not have deterministic effects. It is quite possible that an action attempted by an agent can fail (in our experiments, a failed action leaves the agent in the same state). The results of decreasing action success probability are shown in Figure 3.8. While the model computed from the USM tree departs farther from the optimal



(a) Deterministic actions

(b) 0.1 action noise

(c) 0.2 action noise

(d) 0.3 action noise

(e) Legend for the above graphs

Figure 3.8: Results for the maze in Figure 3.6(c) using various levels of action noise (i.e., 1 – action-success-probability). In all the above graphs, the X-axis is the diminishing sensor accuracy and the Y-axis is the average reward per agent action. The above results are averaged over 10 different executions for each observation accuracy and method. All variances were below 0.015 and in most cases below 0.005.

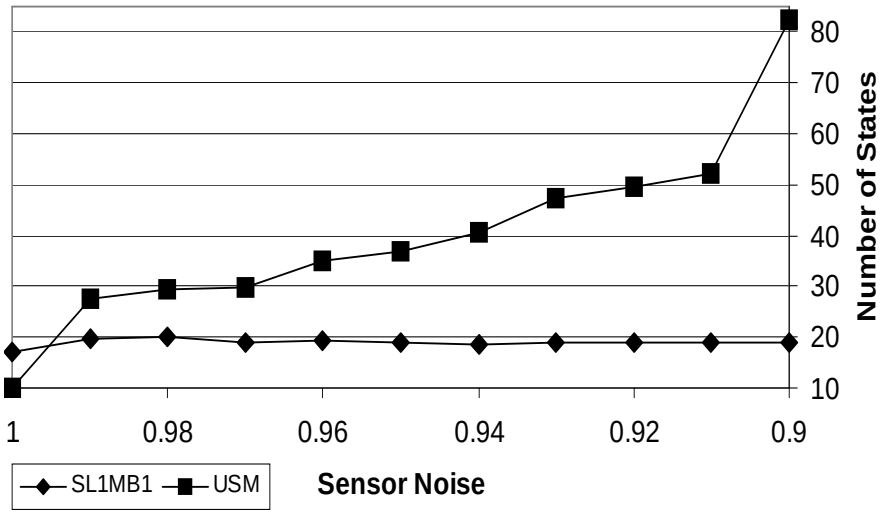
model, it still outperforms USM by approximately the same degree. The USM-based models performance degrade partially due to the increased number of leaves (and hence, states) in the presence of noisy actions, as failed transitions cannot be expressed in a single leaf and instead deeper branches are created for such histories.

The memory bits model in our tests produced smaller state spaces than USM. This is because the memory bits algorithm defines a constant upper bound on the number of states defined by all the possible combinations of the internal memory states and the observations.

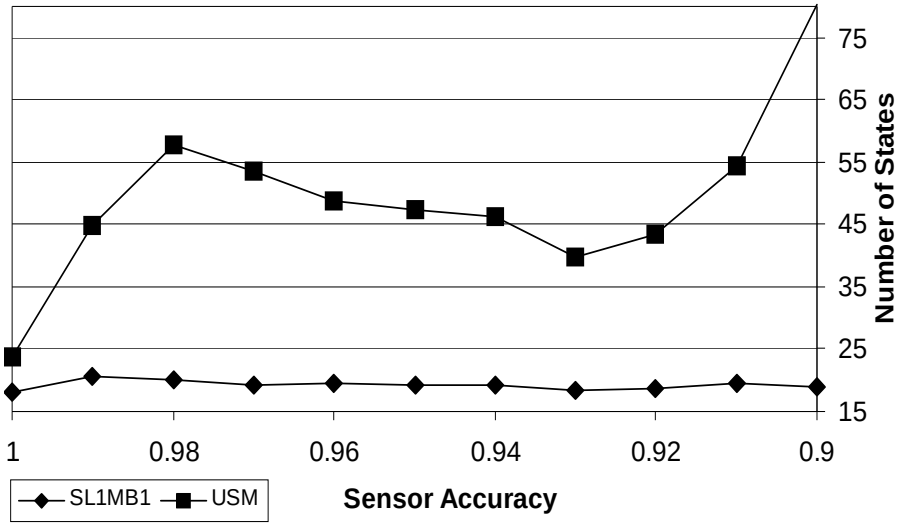
3.5 Model-Based Online Learning[6]

In an online learning process, the agent continuously improves the model definition and policy, as opposed to offline learning where the agent initially learns a complete model and only then computes a policy for the learned model. There are some attractive advantages to learning online. An agent that learns online can provide a policy that improves constantly, as opposed

[6]Parts of this section appeared in (Shani et al., 2005b)



(a) Discovered states on the maze in Figure 3.6(a).



(b) Discovered states on the maze in Figure 3.6(b).

Figure 3.9: Increase in the number of states discovered by Sarsa(λ) using one memory bit and USM as sensor noise increases.

to offline algorithms that present a fixed policy until sufficient data have been gathered. Continuous exploration also allows the agent to cope with slow changes in the environment. It is possible to use an offline algorithm for policy computation by repeatedly executing a policy (that might initially be random) and once a certain quantity of data has been gathered, update the model and compute a new policy. Most algorithms in Section 3.1.7 take this approach. The disadvantages of this approach are:

- The needed quantity of data for updating is unclear. It is difficult to decide when to stop experimenting and update the model and policy. As these computations are extensive, unneeded updates are extremely undesired.

- During the exploration interval it might be that the agent continues to execute actions that were already observed as harmful, since those data have yet to be used by the learning mechanism.
- The length of time the agent needs to update the model and recalculate a policy might require it to stop operating for a while, as most of the algorithms mentioned above are computationally expensive.

In this section we present an algorithm that continually and incrementally updates a model and a policy simultaneously online, resolving the above concerns. The model is based on the USM algorithm for construction and update.

3.5.1 Augmenting USM with a Belief-State

In the previous section we explained how a USM can be transformed into a POMDP, by also learning transition, reward, and observation functions. We want to learn a value function V, composed of α-vectors and maintain it online. A policy π_V is derivable from the value function using $\pi_V(b) = \arg\max_{a:\alpha_a \in V} b \cdot \alpha_a$, where b is a belief vector over the states of the POMDP. Hence, we need to maintain a belief state online.

The calculated belief-state can also be used to add new instances to the USM tree. Instead of adding an instance to a single path as USM does, or into several paths based solely on a sensor model as NUSM does, it is possible to leverage the belief state for insertion.

Instead of referring to a single (possibly noisy) observation, we can consider all possible observations weighted by their probability $p(o|b)$ where b is the current belief state. In USM, $p(o|b)$ is easy to compute as each state (leaf) corresponds to a specific assignment to world features (for example, a specific wall configuration), and therefore $p(o|b) = \sum_{s \in S_o} b(s)$ where S_o is the set of all states that correspond to world configuration o.

We can now insert a new instance T_t into all states, weighted by $p(o|b_t)$, or replace the noisy observation o_t with the observation with maximal probability $\arg\max_o p(o|b_t)$. In the experiments reported below we take the second approach.

Any algorithm that assumes an unknown state space may require a modification of the state space. We assume that the modification is done by splitting a state $s \in S$ into several states to form the new state space S' (as in UDM and USM, for example). Assuming that state s was split, for each state s' in the new state space S' we compute a new belief value:

$$b'(s') = \begin{cases} \frac{b(s)}{|S'-S|} & , \quad s' \in S' - S \\ b(s') & , \quad s' \in S \end{cases} \tag{3.26}$$

meaning that the probability of being in the split state s is equally divided between the new states. The vectors in $\mathcal{V}$ can also be updated by:

$$\alpha'(s') = \begin{cases} \alpha(s) & , \quad s' \in S' - S \\ \alpha(s') & , \quad s' \in S \end{cases} \tag{3.27}$$

for each vector $\alpha \in \mathcal{V}$ and state s' in the new state space S', so that the value of each new state is the same as the value of the state s they were split from.

3.5.2 Approximating POMDP Solution Using a Belief State

As we have seen in Section 2.2.2, there are several approximation techniques for POMDPs that use a solution to the underlying MDP (the MDP that describes the environment) to calculate

an approximation to the POMDP optimal policy. Such methods include the Most Likely State (MLS), Voting, and Q_{MDP}.

Using the computed belief-state and the Q-values we maintain in the leaves, these approximations are immediately usable. These approximation methods are extremely rapid to compute and use online. They all, however, rely on some MDP approximation, and are therefore inferior to a value function computed using a complete POMDP representation, even on the simple domains we experiment with.

3.5.3 Using Iterative Perseus to Maintain a Policy

The Perseus algorithm (Algorithm 11) is executed using a POMDP and a set of belief points. However, since convergence of the algorithm still takes considerable time, we would like to incrementally improve a value function (and hence, a policy) as we learn and act, without requiring the complete execution of Perseus after each step. Our method is an online version of the Perseus algorithm — an algorithm that receives a single belief point and adjusts the computed value function accordingly.

Algorithm 7 is an adaptation of the original algorithm, using two value functions – the current function V and the next function V'. V' is updated until no change has been noted for a period of time, upon which V' becomes the active function V.

Algorithm 7 Iterative Perseus

1: **if** $V'(b) < V(b)$ **then**
2: $\quad \alpha \leftarrow backup(b)$
3: **if** $\alpha \cdot b > V(b)$ **then**
4: $\quad V' \leftarrow V' \cup \{\alpha\}$
5: **else**
6: $\quad V' \leftarrow V' \cup \{\max_{\beta \in V} \beta \cdot b\}$
7: **if** V' has not been updated in a long while **then**
8: $\quad V \leftarrow V'$
9: $\quad V' \leftarrow \phi$

In the original, offline version of Perseus, belief points for updating are selected randomly. The iterative version we suggest selects the points we update following some traversal through the environment. If this traversal is chosen wisely (i.e., using a good exploration policy), we can hope that the points that are updated improve the solution faster.

Using any value function based on α-vectors in conjunction with a model learning algorithm can be problematic, because α-vectors can only represent a lower bound. As the value of the next value function over all (tested) belief points always increases, wrong over-estimates, originating from some unlearned world feature, can persist in the value function even though they can not be achieved. Such maxima can be escaped using some randomization technique, such as occasionally removing vectors, or by slowly decaying older vectors. We note this problem, even though it does not manifest in our experiments.

3.5.4 Performance of The Online POMDP Learning

We conducted a set of experiments running the USM-based POMDP on the mazes in Figure 3.6. During execution the model maintains a belief state and states were updated as explained in Section 3.5.1. The agent executes the iterative Perseus algorithm (Algorithm 7) for each observed belief state. During the learning phase, the next action was selected using the MLS (most likely state) technique (Section 2.2.2). Once the average discounted reward collected by

the algorithms passed a certain threshold, exploration was stopped (as the POMDP policy does not explore).

From this point onwards, learning was halted and the executions were continued for 5000 iterations for each approximation technique to calculate the average discounted reward gained — MLS, Voting, Q_{MDP}, and the policy computed by the iterative Perseus algorithm. In order to provide a gold standard, we manually defined a POMDP model for each of the mazes above, solved it using Perseus and ran the resulting policy for 5000 iterations.

Execution time in our tests was around 6 milliseconds for an iteration of USM, compared to about 234 milliseconds for an iteration of the POMDP learning (including parameter and policy updates) on the maze in Figure 3.6(b) with sensor accuracy 0.9, on a Pentium 4 with 2.4 GHz CPU and 512 MB memory.

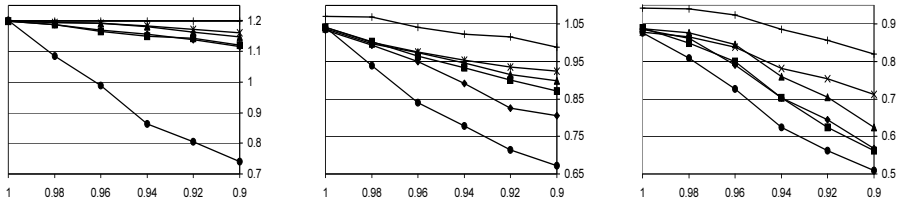
The POMDP learning algorithm is much slower than the original USM but still feasible for online robotic application, where an action execution is usually measured in seconds. Moreover, much of that time is required for simply updating the belief state, an operation required even for only executing a POMDP policy. For example, the executed policy of the manually defined model (without any learning), takes about 42 milliseconds per iteration.

Our experimental results are presented in Figure 3.10. The graphs compare the performance of the original USM algorithm and our various enhancements: the belief state approximations (MLS, Voting, and Q_{MDP}) and the policy computed by the Iterative Perseus algorithm (Algorithm 7), denoted Policy. We also show the results of the policy for the manually defined model, denoted Optimal Model as an upper bound.

Observe that the performance of USM decreases sharply as observation noise increases, but the performance of the POMDP based methods remains reasonably high. The improvement is due to the fact that all the POMDP methods model the noise using the belief state, whereas pure USM ignores it. The differences between the POMDP solution methods are not too significant for the first two mazes, and are more noticeable in the last model. Iterative Perseus provides better solutions than the approximations in all the experiments. The third model exhibits more uncertainty in the belief states, and a more pronounced reward variability due to error (this maze is less forgiving with respect to deviations from the optimal, especially in the states where the agent observes no walls, where the same action causes a large reward in one state, and a large penalty in the other), making the difference in performance significant. Here, the MDP based methods (MLS, Voting, and Q_{MDP}) do not perform nearly as well as the computed policy on that model.

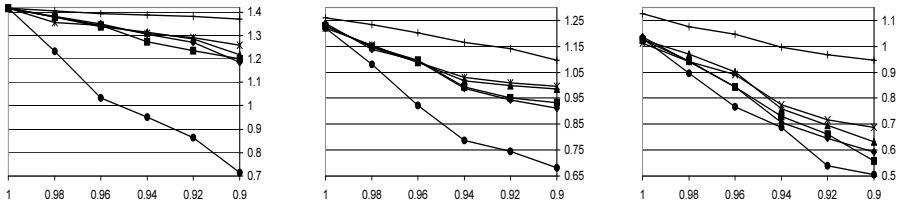
We can expect that as the complexity of the model and the reward variances increase, the advantage of the policy computed by the incremental algorithm should become apparent. The performance of the iterative Perseus policy is still not as good as the policy computed by the manually defined model. This is because the model defined by USM contains many unnecessary states, since we associate a state with every leaf of the tree constructed by USM. World states that can be reached in more than one way may result in multiple states in our induced model. Naturally, the policy for a less accurate model is likely to be less effective in the real environment. However, we report that experiments on identical models (that we do not show here) demonstrate that the policy resulting from our incremental version of Perseus performs as well as the policy resulting from running the standard (offline) version of Perseus.

The performance of the policy generated from the USM-based model is not as good as the policy of the manually defined model. This is because the USM-based model has many redundant states, as explained in Section 3.1.6. The lower performance is not due to the use of the iterative Perseus instead of the offline version. We also executed a simulation on a predefined POMDP model, using iterative Perseus to compute a policy. The resulting policy was no worse than the one computed by the offline Perseus on an identical model.



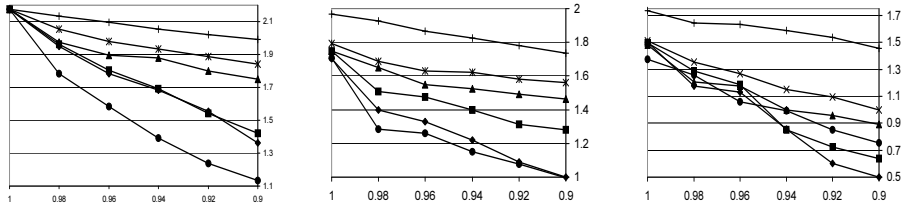
(a) Action noise 0.0. (b) Action noise 0.1. (c) Action noise 0.2.

Results for the maze in Figure 3.6(a).



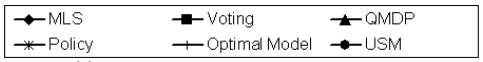
(d) Action noise 0.0. (e) Action noise 0.1. (f) Action noise 0.2.

Results for the maze in Figure 3.6(b).



(g) Action noise 0.0. (h) Action noise 0.1. (i) Action noise 0.2.

Results for the maze in Figure 3.6(c).



(j) Legend for the above graphs

Figure 3.10: Results for the mazes in Figure 3.6. In all the above graphs, the X axis is the diminishing sensor accuracy p, and the Y axis is the average discounted reward per agent action. The above results are averaged over 5 different executions for each observation accuracy and method. All variances were below 0.01 and in most cases below 0.005.

Chapter 4

Scaling Point-Based Solvers

Learning a POMDP model is only the first step towards solving the reinforcement learning problem. The second part is the computation of an optimal or approximate policy. In the past, it was only possible to compute policies for very small problems using exact methods. However, in the past few years there has been a considerable step forward towards scaling up to real world domains, mainly due to point-based methods for solving POMDPs.

We begin this chapter with an introduction to point-based methods, discussing most of the important algorithms of this family. We then present our own contributions to scaling up point-based solvers. We first present a discussion over the importance of the order by which value function updates are executed, then move on to discuss a new trial-based algorithm that is extremely fast in computing good trajectories.

We end this chapter with a thorough overview of point-based algorithms in factored POMDPs using Algebraic Decision Diagrams (ADDs). We show how to implement all the required operations for a point-based algorithm using ADDs and then discuss several improvements that allow us to scale up to POMDPs with millions of states.

4.1 Point-Based Solvers

The Incremental Pruning algorithm computes the value function at time $t-1$ by computing all the possible single step backups of the value function at time t. It then prunes out vectors that do not participate in the upper envelope. However, the Bellman update for POMDPs (Equation 2.13) is written for a single belief state.

Indeed, some researchers suggested executing updates for specific belief states. (Cheng, 1988) suggested finding the belief points where the α-vectors interact and computing new α-vectors for these belief points. (Cassandra et al., 1997) suggest adding to the full backup process of the Incremental Pruning algorithm local improvements using updates over specific belief states.

A separate form of value iteration for POMDPs — the grid based approach (Lovejoy, 1991; Brafman, 1997; Hauskrecht, 2000; Zhou & Hansen, 2001) — defines the value function through direct mapping from belief points to values. The projection operator computes the value of a belief point that is not directly mapped in the value function. This operator is implemented differently by each grid-based approach. The grid-based family of algorithms always updates the value of specific points.

The value function update rule:

$$V_{n+1}(b) = \max_a [b \cdot r_a + \gamma \sum_o pr(o|a,b) V_n(\tau(b,a,o))] \tag{4.1}$$

where $r_a(s) = R(s,a)$, can be rewritten (Cassandra et al., 1997; Pineau et al., 2003; Spaan & Vlassis, 2005):

$$
\begin{aligned}
V_{n+1}(b) &= \max_a [b \cdot r_a + \gamma \sum_o pr(o|a,b) V_n(\tau(b,a,o))] && (4.2)\\
&= \max_a [b \cdot r_a + \gamma \sum_o pr(o|a,b) \max_{\alpha \in V_n} \sum_{s'} \tau(b,a,o)(s')\alpha(s')] && (4.3)\\
&= \max_a [b \cdot r_a + \gamma \sum_o \max_{\alpha \in V_n} \sum_{s'} O(a,s',o) \sum_s tr(s,a,s') b(s) \alpha(s')] && (4.4)\\
&= \max_a [b \cdot r_a + \gamma \sum_o \max_{\alpha \in V_n} b \cdot g^{\alpha}_{a,o}] && (4.5)\\
g^{\alpha}_{a,o}(s) &= \sum_{s'} O(a,s',o) tr(s,a,s') \alpha(s') && (4.6)
\end{aligned}
$$

The equations above can be rewritten into vector notation to compute a new α-vector for the specific belief state b:

$$backup(b) = \arg\max_{g^b_a : a \in A} b \cdot g^b_a \quad (4.7)$$

$$g^b_a = r_a + \gamma \sum_o \arg\max_{g^{\alpha}_{a,o} : \alpha \in V} b \cdot g^{\alpha}_{a,o} \quad (4.8)$$

Pineau, Gordon, and Thrun (Pineau et al., 2003) suggest to stop using complete value function updates at all and use only point-based backups. This idea was also shared by the grid-based family, but Pineau et al. also suggested using only a finite set of reachable belief points for this process.

Point-based algorithms therefore:

1. Compute a value function defined as a set of α-vectors.
2. Improve the value function using point-based backups (Equation 4.7).
3. Execute backups over a finite set of reachable belief points.

Algorithms of this family differ on two main scales:

- The choice of the finite subset of belief points.
- The execution order of point-based backups over the selected belief points.

We shall now review the important point-based algorithms, focusing on the above two scales.

4.1.1 Point-Based Value Iteration (PBVI)

The first algorithm of this family, Point Based Value Iteration (PBVI — Algorithm 8) was introduced by (Pineau et al., 2003). At each iteration, PBVI adds points as far as possible from the current points in the belief set B. Thus, PBIV attempts to cover the space of reachable belief states as best as possible. In the limit, the entire reachable belief space will be covered and the value function will be exact.

A major deficiency with such a selection is that it does not take into consideration the location of the rewards or goals of the environments. As such, if a reward is reached only after a large number of steps from the initial state, PBVI will spend many iterations until a belief state that has non-zero probability of receiving that reward is added to B.

Algorithm 8 PBVI

1: $B \leftarrow \{b_0\}$
2: **while** true **do**
3: $\quad Improve(V, B)$
4: $\quad B \leftarrow Expand(B)$

Algorithm 9 Improve(V, B)

Input: V — a value function
Input: B — a set of belief points
1: **repeat**
2: $\quad$ **for each** $b \in B$ **do**
3: $\quad\quad \alpha \leftarrow backup(b)$
4: $\quad\quad add(V, \alpha)$
5: **until** V has converged

(Pineau, Gordon, & Thrun, 2006) and (Izadi, Precup, & Azar, 2006) acknowledge this deficiency and attempt to fix it by selecting actions based on the current value function. However, these techniques still do not find a heuristic that is capable of finding far rewards. Also, all the suggested heuristics for the expansion of B are computationally expensive. Thus, PBVI and its variations did not show the ability to scale up to domains with more than 1000 states.

The value function improvement is done in PBVI and its variants using an arbitrary order of backups. In the first few steps this update phase is not too expansive but later, as the number of belief states in B increases, such an update becomes very inefficient. This update also may introduce many redundant α-vectors that may be avoided, and thus slow down the point-based backups.

4.1.2 Perseus

The Perseus algorithm (Spaan & Vlassis, 2005) attempts to speed up the value iteration process by avoiding the two main sources of intractability in PBVI — the heuristic for expanding B and the large number of α-vectors.

Perseus selects the belief point set B using a random walk through the belief space. This method for gathering belief states is very fast even if the needed number of belief points is large. No filtering is done of repeating belief points, under the assumption that belief points that are observed often during a random walk also need to be updated more often.

Given the gathered set of belief points B, Perseus (Algorithm 11) selects the next belief point b to update in random. A new α-vector is computed for b. Then, b and all other belief points in B that were improved using the new α-vector, are removed from the belief point set and thus no longer considered for update. Once there are no longer any points in the belief point set, all points in B are reintroduced and a new iteration begins.

Algorithm 10 Expand(B)

Input: B — a set of belief points
1: $B' \leftarrow B$
2: **for each** $b \in B$ **do**
3: $\quad Successors(b) \leftarrow \{b' | \exists a, \exists o\ b' = \tau(b, a, o)\}$
4: $\quad B' \leftarrow B' \cup argmax_{b' \in Successors(b)} dist(B, b')$
5: **return** B'

Algorithm 11 Perseus

Input: B — a set of belief points

```
repeat
    B̃ ← B
    while B̃ ≠ φ do
        Choose b ∈ B̃
        α ← backup(b)
        if α · b ≥ V(b) then
            B̃ ← {b ∈ B̃ : α · b < V(b)}
        add(V, α)
until V has converged
```

For small scale problems Perseus performs very well. It computes good value functions with a small number of vectors. Unfortunately, for larger domains Perseus has several shortcomings. First, in domains that require a complicated sequence of actions to achieve a goal, it is unlikely that a random walk would find that sequence. Random walks that are long enough to achieve such goals are also likely to collect a large number of unheeded points, which slows down the value function computation. Second, in complex domains, where the order of backups is important, using backups in a random order may require a large number of backups until the value function converges. Also, in larger domains, belief states may become memory consuming and maintaining a large number of belief states in memory may become impossible. Finally, as opposed to PBVI, Perseus does not guarantee an optimal value function.

4.1.3 Heuristic Search Value Iteration (HSVI)

The first attempt at a trial-based algorithm for POMDPs was suggested by (Bonet & Gefner, 1998). RTDP-BEL creates a discretization of the belief states by allowing only belief states with entries of structure m/n. Thus the number of possible belief states is reduced, but the coverage of the belief space can be as dense as possible.

RTDP-BEL than executes trials in the same manner as RTDP. For each observed belief state b its discretized closest neighbor b' is updated. RTDP-BEL maintains Q values for each such discretized belief point. In experiments done by us, RTDP-BEL failed to solve even very small problems. As n increases, the number of discretized belief points grows and the algorithm needs many trials to update every point enough times. Moreover, the computed value function (not using any vectors), scales poorly.

(Smith & Simmons, 2004, 2005) suggest a more advanced trial-based algorithm. They use an upper and lower bound over the optimal value function. The upper bound is represented using a direct mapping from belief states to values, following (Hauskrecht, 2000). The lower bound is represented using α-vectors. Heuristic Search Value Iteration (HSVI — Algorithm 12) traverses the belief space using a heuristic designed to reduce the gap between the bounds.

Algorithm 12 HSVI

```
Initialize V̲ and V̄
while V̄(b0) − V̲(b0) > ε do
    Explore(b0, V̲, V̄)
```

Smith and Simmons show that within a polynomial number of updates, the gap between bounds over the initial belief state b_0 will drop beneath ϵ. The heuristic of HSVI is in fact very good and finds good trajectories. It can also be shown that HSVI always converges to

Algorithm 13 Explore(b, $\underline{V}, \bar{V}$)

Input: a belief state b, upper and lower bounds on the value function $\underline{V}, \bar{V}$.

1: **if** $\bar{V}(b) - \underline{V}(b) > \epsilon\gamma^{-t}$ **then**
2: $a^* \leftarrow \arg\max_a Q_{\bar{V}}(b, a')$
3: $o^* \leftarrow \arg\max_o(pr(o|b, a^*)(\bar{V}(\tau(b, a, o)) - \underline{V}(\tau(b, a, o))))$
4: $Explore(\tau(b, a^*, o^*), \underline{V}, \bar{V})$
5: $add(\underline{V}, backup(b, \underline{V}))$
6: $\bar{V} \leftarrow HV(b)$

the optimal policy. Another attractive feature of trial-based algorithms is that they need not remember a large number of belief points — only the points in the current trajectory.

On the other hand, updating HSVI upper bound is extremely time consuming and it appears that HSVI spends most of its time on upper bound updates. While the upper bound is useful for the traversal heuristic it does not participate in the final solution (value function). Also, the need to maintain all the points in the upper bound eliminates the benefit of trial-based algorithms — the small number of maintained points.

HSVI executes point-based backups over belief states in reversed order. This order provides a significant contribution to the speed of convergence. As the belief state value "trickles down" from successor to predecessor, updating them in a reversed order ensures that the new value of the successor is considered when updating the value of the predecessor.

4.1.4 Stopping Criteria for Point-Based Methods

The above point-based methods are specified as an anytime algorithm — the algorithm is executed indefinitely and constantly maintains a solution. Nevertheless, some of the algorithms have convergence properties.

Pineau et al. show that the distance between the current value function and the optimal value function can be expressed as a function of the density of the current belief set B. That is, if the maximal distance between points in B and any reachable other point is small, than the value function is close to convergence. Measuring the distances between any point in the reachable belief space is difficult, but this distance is always less than the internal distances between points in B, and distances between points in B and the corner of the belief simplex (deterministic belief points). Therefore, for each ϵ, we can compute whether the current policy is ϵ close to convergence by looking at the current belief set B.

Smith and Simmons show their HSVI algorithm to always reduce the gap between the bounds. Given the two bounds, we can always compute the accuracy of the value function by observing the gap over the initial belief state. To compute a value function that ϵ close to optimal we can wait until the gap over b_0 is less than ϵ. We can hence use the gap over b_0 as a stopping criterion.

Perseus does not suggest such convergence guarantees. First, since the belief points were gathered using a random walk it is possible that some important points were never encountered. Therefore, we cannot guarantee the quality of the resulting value function. It is also unclear when can we stop Perseus. It is possible that a complete iteration over B resulted in an ϵ improvement, yet an additional iteration will improve the value function by more than ϵ.

4.1.5 Online Methods

Another promising approach for scaling up is an online search in belief space (Washington, 1997; Paquet, Tobin, & Chaib-draa, 2005). When the agent is at belief state b it can expand

the belief space starting from b until a desirable belief state has been reached. This search tree creates an AND-OR tree (or graph), where in an observation node the search must continue to all children, and in an action node the search can select which action child to expand. The search can be improved by using an upper and lower bound over the value function, allowing the pruning of dominated tree branches. (Ross & Chaib-draa, 2007) make the search faster by reusing parts of the search tree that were traversed in the previous iteration.

These methods appear to provide good policies rapidly in a number of large scale benchmarks in literature. It would be interesting to combine the online search with the computation of a policy, such that after a while there is no longer any need for additional search and the learned policy can be used instead. Such an approach will be highly useful for agents which cannot spend a long time pre-computing a policy.

4.2 Prioritized Point Based Value Iteration[1]

Point-based algorithms compute a value function using α vectors by iterating over some finite set of belief points and executing a sequence of backup operations over these belief points. For each set of belief points there are many possible sequences of backup executions. As our goal is to approximate the value function as quickly as possible, we say that a backup sequence seq_1 is better than sequence seq_2 if seq_1 is shorter than seq_2, and produces a policy which is no worse than the one produced by seq_2.

A better sequence does not always mean that the algorithm that uses it will be more efficient. It is possible that an algorithm executes a good sequence, but the time to compute it is much more than the execution of a worse sequence. Ideally we would like to obtain a good sequence rapidly.

We suggest creating (hopefully) better sequences using a heuristic that predicts useful backups. Clearly, the heuristic must be efficiently computable, so that the overhead of computing the heuristic does not outweigh any savings achieved by performing fewer backups.

4.2.1 Prioritizing MDP Solvers

A comparable scheme used for prioritizing in MDP solvers, suggests performing the next backup on the MDP state that maximizes the Bellman error:

$$e(s) = max_a[R(s,a) + \sum_{s'} tr(s,a,s')V(s')] - V(s). \tag{4.9}$$

$e(s)$ measures the change in the value of s from preforming a backup. (Wingate & Seppi, 2005) present a very simple version of value iteration for MDPs using prioritization (Algorithm 14).

Algorithm 14 Prioritized Value Iteration for MDPs

1: $\forall s \in S, V(s) \leftarrow 0$
2: **while** V has not converged **do**
3: $\quad s \leftarrow \arg\max_{s' \in S} e(s')$
4: $\quad backup(s')$

A key observation for the efficiency of their algorithm is that after a backup operation for state s, the Bellman error recomputation need be performed only for the predecessors of s $\{s' : \exists a, tr(s', a, s) \neq 0\}$. Hence, after initially setting $e(s) = \max_a R(s,a)$ for each $s \in S$, we update the priorities only for predecessors, avoiding a complete iteration through the state space.

[1] Parts of this section appeared in (Shani et al., 2005a)

4.2.2 Prioritizing POMDP Solvers

While the Bellman error generalizes well to POMDPs:

$$e(b) = max_a[r_a \cdot b + \sum_o pr(o|a,b)V(\tau(b,a,o))] - V(b) \tag{4.10}$$

there are two key differences between applying priorities to MDPs and POMDPs.

First, a backup update affects more than a single state. A new vector usually improves the local neighborhood of its witness belief point, but may improve the value for the entire belief space. As such, both the current value of any belief state may change following a backup operation, and the value of one of its successors, and hence the error $e(b)$ may also change due to the new vector.

Second, the set of predecessors of a belief state cannot be efficiently computed, and its size is potentially unbounded. Consider, for example, a case where in some state s the agent receives a unique observation o. Given prior belief state b, after observing o the next belief state $b' = \tau(b, *, o)$ must be such that $b'(s) = 1.0$.

Given these two problems it is not possible to update priorities only for the set of predecessors of a belief state whose value has just improved.

Moreover, even supposing that some similarity metric for finding the neighborhood of a belief point were defined, and that computation of the predecessor set were only for the finite set of belief points we use, directly applying the approach would still not be worthwhile. In practice, algorithms such as Perseus frequently converge to an optimal solution while computing fewer backups than the number of belief points in the finite set. Pre-computations such as similarity matrices will take more time than the original algorithm they are designed to improve in the first place.

As we cannot find the set of belief states affected by the backup operation directly, we recompute the Bellman error for all belief states after every backup from scratch. When the number of belief points we use is relatively small, this computation can be done without seriously damaging the performance. As the size of the problem — states, actions, observations and belief set size — increases, we can no longer afford the overhead of recomputing the Bellman error for all belief states.

We therefore take a stochastic approach, sampling a subset of the belief points set and computing the Bellman error only for this sampled subset. If the sampled subset does not contain a point with positive error, we sample again from the remaining subset (without repetitions) until a belief point with positive error is found. If there is no belief point with a positive Bellman error then the value function has reached a fixed point and has therefore converged to an optimal solution over the finite set of belief points.

While the upper bound complexity of both the backup operation and the error computation is identical, $O(|A||\Omega||S|^2)$, in practice, computing belief updates (Equation 2.7) is much faster than the computation of $g^{\alpha}_{a,o}$ (Equation 4.6). This is because belief states usually have many zero entries compared to α-vectors. Using data structures that support maintaining and iterating only over non-zero entries, all operations above can be implemented efficiently.

4.2.3 Prioritizing Existing Algorithms

We first show how prioritization can be used to enhance the performance of current algorithms. We suggest replacing the backup selection mechanism of existing algorithms with a prioritization scheme.

Prioritizing Perseus is straight forward. The chosen step (Algorithm 11, line 4) is implemented in Perseus as a uniform selection among any of the current belief points inside $\tilde{B}$.

Prioritized Perseus uses the Bellman error computation to choose a belief point whose value can be improved the most. As a result, backups are executed over belief states in order of reduced priorities. These priorities are updated after each backup, but belief points who were already improved in the current iteration are no longer considered.

PBVI improves its value function (Algorithm 8, line 3) by arbitrarily passing over all belief points and performing backup executions. We replace this inefficient computation of the 'Improve' operation with our PVI algorithm. As the number of points used by PBVI is relatively small, we did not use sampling when computing the Bellman error.

4.2.4 Prioritized Value Iteration

Finally, we present an independent algorithm — Prioritized Value Iteration (PVI). Like Perseus, PVI computes a value function over a fixed set of belief points collected before the algorithm is executed. However, Perseus operates in iterations over the set of belief points, attempting to improve all belief points between considering the same belief state twice. PVI considers at each step every possible belief state for improvement. It is likely, therefore, that some belief states will be backed up many times, while other belief states will never be used.

Algorithm 15 presents our PVI algorithm. The algorithm described here is the clean version of PVI, but in practice we implement the *argmax* operation (line 2) using our sampling technique (Algorithm 16). Note that if the algorithm is unable to find a belief state b with non-zero error (Choose returns *nil*), then the value function over B has converged.

If the prioritization metric is good, PVI executes a shorter sequence of backup operations. Indeed, experiments show that it uses significantly fewer backup operations than Perseus using our locally greedy Bellman error prioritization metric.

Algorithm 15 Prioritized Value Iteration

Input: B — a set of belief points

1: **while** V has not converged **do**
2: $b^* \leftarrow \arg\max_{b \in B} e(b)$
3: $\alpha \leftarrow backup(b^*)$
4: $add(V, \alpha)$

Algorithm 16 Choose

Input: B — a set of belief points, k — sample size

1: $B' \leftarrow B$
2: $b_{max} \leftarrow nil$
3: **while** B' not empty **do**
4: **for** $i = 0$ to k **do**
5: Select b with uniform distribution from B' and remove it
6: **if** $e(b) > e(b_{max})$ **then**
7: $b_{max} \leftarrow b$
8: **if** $e(b_{max}) > 0$ **then**
9: **return** b_{max}
10: **return** *nil*

4.2.5 Gathering Belief Points Through Heuristic Search

A second problem with the Perseus algorithm is the use of a random walk to gather the set B of belief points. In small and medium sized domains it is possible to reach all interesting belief

points through a random walk. In larger or more complex domains, however, it is unlikely that a random walk would visit every location where a reward can be obtained.

In the case where a sequence of actions is required to obtain a reward, while every deviation from the sequence causes the system to reset to its original state, it is unlikely that a random walk will be able to find the sequence.

Consider, for example, the float-reset problem, where n states are connected in a chain. The system has 2 actions: float, which moves the agent with equal probability either up or down the chain, and reset, which sends the agent to the initial state. The agent receives a reward for executing reset at the last state in the chain. In the last state in the chain the agent receives a special observation. A good policy would be to float until the special observation is perceived and then activate the reset action. A random walk would take a very long time until such a specific sequence of actions is executed.

We suggest replacing the random walk of Perseus and PVI with a heuristic search, based on the Q_{MDP} policy. The Q_{MDP} policy uses the optimal Q-function Q^* of the underlying MDP to define a Q-function over the POMDP belief space:

$$Q_{MDP}(b, a) = \sum_s b(s)Q^*(s, a) \tag{4.11}$$

The POMDP policy is then defined by:

$$\pi_{Q_{MDP}}(b) = \arg\max_a Q_{MDP}(b, a) \tag{4.12}$$

A well-known problem with MDP-based heuristics for POMDP models is that MDP policies do not estimate the value of observations. We overcome this difficulty by using an ϵ-greedy exploration heuristic. The heuristic search allows us to use smaller, more focused sets of belief points and, thus, to reduce the runtime of our algorithms.

4.2.6 Empirical Evaluations

Heuristic Search

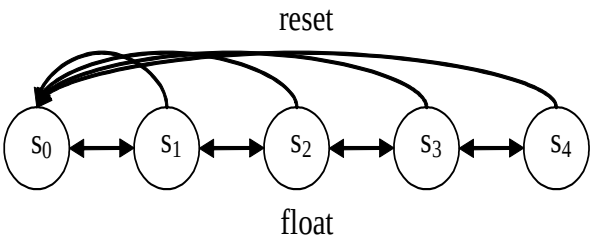
We would first like to evaluate the benefits of our heuristic search method for gathering the belief set B over the random walk used by Perseus.

The number of steps a random walk requires to find the reward of the float-reset problem is shown in Figure 4.1. A heuristic search very rapidly found the reward; with $n = 10$, all 10 runs of length 100 found the reward, with $n = 5$, all 10 runs of length 20 found the reward.

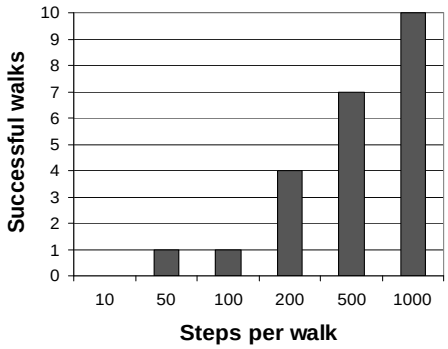
In other domains the agent may receive multiple rewards in different places. A random walk would need to visit all these places in order to allow Perseus to include these rewards in the POMDP value function. We experimented with two instances of the RockSample domain (Smith & Simmons, 2005), both with an 8×8 board, one with 4 rocks and one with 6 rocks. Sampling every rock results in a different reward. The belief set gathering process must pass through the locations of the rocks on the board. The effect that the size of the belief set over the number of rocks visited during the gathering process and hence, the ADR, is shown in Figure 4.2.

Improved Evaluation Metrics

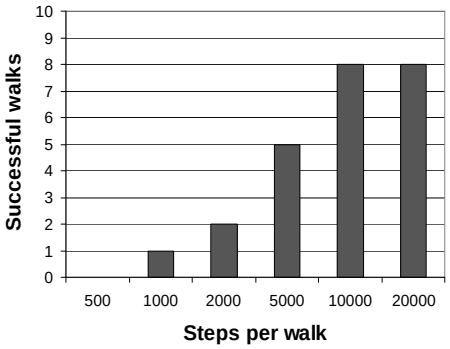
Previous researchers (Brafman, 1997; Pineau et al., 2003; Paquet et al., 2005; Smith & Simmons, 2005; Spaan & Vlassis, 2005) limited their reported results to execution time, average discounted reward, and, in some cases, the number of vectors in the resulting value function.



(a) The Float reset problem with $n = 5$ states.

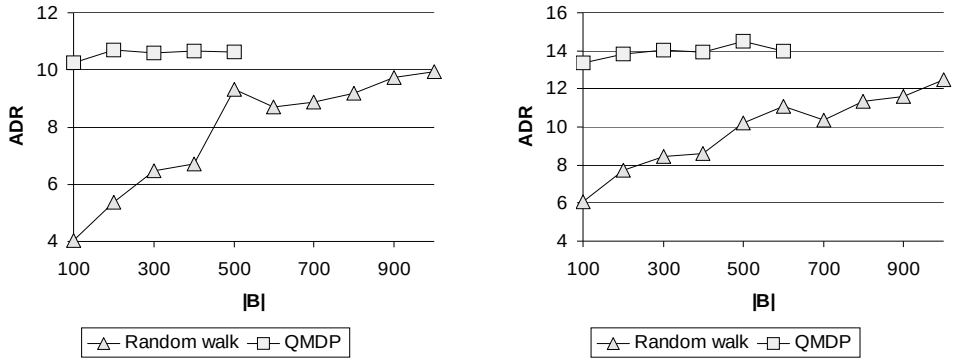


(b) Results for float reset with 5 states.



(c) Results for float reset with 5 states.

Figure 4.1: Number of times, out of 10 runs, a random walk succeeds in finding the goal within the specified number of steps over the float-rest problem with $n = 5$ (a) and $n = 10$ (b).



(a) Results for RockSample 8×8 and 4 rocks. (b) Results for RockSample 8×8 and 6 rocks.

Figure 4.2: ADR computed by Perseus as a function of the belief set size over the RockSample problem with an 8×8 board and 4 rocks (a) or 6 rocks (b). Belief sets are computed by a random walk or the Q_{MDP} heuristic.

Value function evaluation — Average discounted reward (ADR) is computed by simulating the agent interaction with the environment over a number of steps (called a *trial*) and

averaging over a number of different trials:

$$\frac{\sum_{i=0}^{\#trials} \sum_{j=0}^{\#steps} \gamma^j r_j}{\#trials} \tag{4.13}$$

ADR is widely agreed to be a good evaluation of the quality of a value function.

ADR can, however, present very noisy results when the number of trials or the number of steps is too small. For example, on the Hallway example, with 250 trials per ADR, we observed a noise of about 0.5 (about 10% of the optimal performance), while with 10,000 trials the noise dropped to around 0.0015.

In our experiments we interrupted the executed algorithms occasionally to check the ADR of the current value function. We have observed that, in some cases, an algorithm managed to produce a value function providing surprisingly good ADR, but additional backups caused a degradation in ADR. We treat such cases as a noise in the convergence of the value function. To remove such noise we used a first order filter with weight 0.5. The algorithm was stopped once the filtered ADR had converged to a predefined target.

Execution time — It was previously observed that execution time is a poor estimate of the performance of an algorithm. Execution time is subject to many parameters irrelevant to the algorithm itself, such as the machine and platform used to execute it, the programming language, and the level of implementation. It is also important to report CPU time rather than wall-clock time.

As all algorithms discussed in this paper compute a value function using identical operations such as backups, τ function computations, and inner products ($\alpha \cdot b$), it seems that recording the number of executions of those basic building blocks of the algorithm is more informative.

Backup operations themselves do not make a good estimator because they depend on the number of vectors in the current value function. A better estimation is hence the $g_{a,o}^{\alpha}$ computation (Equation 4.6), which depends only on the system dynamics.

Memory — While the size of the computed value function is a good estimate for the execution time of the resulting policy, it can also be used to estimate the memory capacity required for the computation of the algorithm.

A second indication for the amount of required memory is the number of maintained belief points throughout the execution of the algorithm. As some operations (e.g., the Bellman error computation) can be highly improved when caching more belief states, we also report the number of belief states an algorithm uses.

Experimental Setup

In order to test our prioritized approach, we tested all algorithms on the full set of standard benchmarks from the point-based literature: Hallway, Hallway2 (Littman et al., 1995b), TagAvoid (Pineau et al., 2003), and RockSample (Smith & Simmons, 2004). Table 4.1 contains the problem measurements for the benchmarks including the size of the state space, action space and observation space, the number of belief points in the set $|B|$ used for Perseus, Prioritized Perseus and PVI, and the error in measuring the ADR over 10,000 trials for each problem.

We implemented in Java a standard framework that incorporated all the basic operators used by all algorithms such as vector inner products, backup operations, and τ function. All reported results were gathered by executing the algorithms on identical machines —- $x86$ 64-bit machines, dual-proc, processor speed 2.6Ghz, 4Gb memory, 2Mb cache, running Linux and JRE 1.5.

As previous researchers have already shown the maximal ADR achievable by their methods, we focus our attention on convergence speed of the value function to the reported ADR. We

Problem	$\|S\|$	$\|A\|$	$\|O\|$	$\|B\|$	ADR Error
Hallway	61	5	21	250	± 0.0015
Hallway2	93	5	17	300	± 0.004
Tag Avoid	870	5	30	350	± 0.045
Rock Sample 4,4	257	9	2	500	± 0.075
Rock Sample 5,5	801	10	2	500	± 0.3
Rock Sample 5,7	3201	12	2	500	± 0.25

Table 4.1: Benchmark domains parameters

executed all algorithms, interrupting them from time to time in order to compute the efficiency of the current value function using ADR over 5000 trials. Once the filtered ADR has reached the same level as reported in past publications execution was stopped. The reported ADR was then measured over additional $10,000$ trials (error in measurement is reported in Table 4.1).

For algorithms that require a given set of belief states B — Perseus, Prioritized Perseus, and PVI — we pre-computed 5 different sets of belief points for each problem. Each belief points set was computed by simulating an interaction with the system following the Q_{MDP} policy with an ϵ-greedy exploration factor ($\epsilon = 0.1$). For each such belief point set we ran 5 executions with different random seeds resulting in 25 different runs for each stochastic method. The number of belief points used for each problem is specified in Table 4.1. Using the Q_{MDP} heuristic for gathering belief points allowed us to use a considerably smaller belief set than (Spaan & Vlassis, 2005).

Algorithms that are deterministic by nature — PBVI, Prioritized PBVI, and HSVI — were executed once per problem.

Results

Our experimental results are presented in Table **??**. For each problem and method we report:

1. Resulting ADR
2. Size of the final value function ($|V|$)
3. CPU time until convergence
4. The number of backups
5. The number of $g^{\alpha}_{a,o}$ operations
6. The number of computed belief states
7. the number of τ function computations
8. The number of inner product operations.

The reported numbers do not include the repeated expensive computation of the ADR, or the initialization time (identical for all algorithms). Results for algorithms that require a pre-computed belief space do not include the effort needed for this pre-computation. We note, however, that it took only a few seconds (less than 3) to compute the belief subset B over all problems.

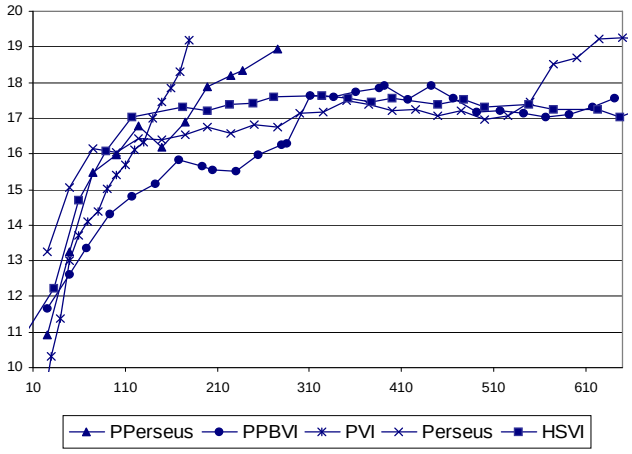
To better illustrate the convergence of the algorithms we have also plotted in Figure 4.3 the convergence of the ADR vs. the number of backups an algorithm performs. The graphs contain data collected over separate executions with fewer trials (500 instead of 10000) so Table **??** has more accurate results.

HSVI is the only method that also maintains an upper bound over the value function ($\bar{V}$). Table **??** contains additional measurements for the computation of the upper bound: the number of points in $\bar{V}$, the number of projections of other points onto the upper bound, and the number of upper bound updates ($HV(b)$ — Equation 2.13).

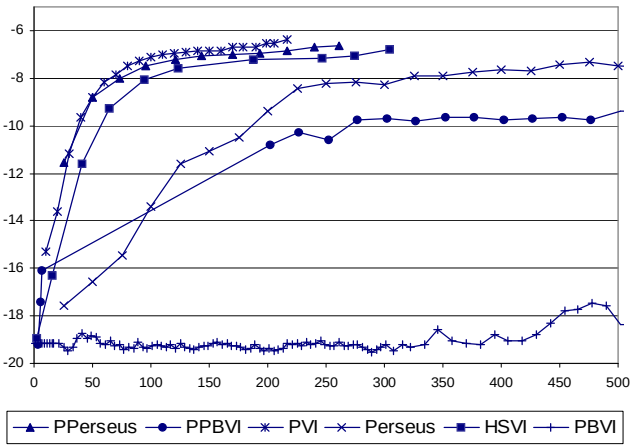
For the stochastic methods we show standard deviations over the 25 runs for all categories.

Method	ADR	$\|V\|$	Time (secs)	#Backups	$\#g^{\alpha}_{a,o}\times 10^6$	#belief states$\times 10^4$
Hallway						
PVI	0.517±0.0027	144±32	**75**±32	**504**±107	3.87±1.75	1.99±0.04
PPerseus	0.517±0.0025	173±43	126±47	607±166	5.52±2.95	1.99±0.04
Perseus	0.517±0.0024	466±164	125±110	1456±388	31.56±27.03	0.03±0
PPBVI	0.519	235	95	725	9.09	1.49
PBVI	0.517	253	118	3959	31.49	1.49
HSVI	0.516	182	314	634	5.85	3.4
Hallway2						
PVI	0.344±0.0037	234±32	75±20	262±43	2.59±0.84	2.49±0.11
Pperseus	0.346±0.0036	273±116	219±155	343±173	4.76±5.48	2.49±0.11
Perseus	0.344±0.0034	578±95	134±48	703±120	17.03±6.08	0.03±0
PPBVI	0.347	109	**59**	**137**	0.61	2.03
PBVI	0.345	128	76	1279	7.96	1.52
HSVI	0.341	172	99	217	1.56	2.11
Tag Avoid						
PVI	-6.467±0.19	204±38	**40**±12	211±38	0.42±0.19	0.16±0
PPerseus	-6.387±0.18	260±43	105±26	265±44	5.27±1.8	1.73±0.02
Perseus	-6.525±0.20	365±69	212±174	11242±10433	28.69±32.09	0.04±0
PPBVI	-6.271	167	50	**168**	2.09	0.41
PBVI	-6.6	179	1075	21708	407.04	0.41
HSVI	-6.418	100	52	304	0.5	0.29
Rock Sample 4,4						
PVI	17.725±0.32	231±41	**4**±2	232±42	0.36±0.14	0.41±0.01
PPerseus	17.574±0.35	229±26	5±2	228±27	0.34±0.08	0.41±0.01
Perseus	16.843±0.18	193±24	158±33	24772±5133	59.96±13.06	0.05±0
PPBVI	18.036	256	229	265	0.62	2.43
PBVI	18.036	179	442	52190	113.16	1.24
HSVI	18.036	123	**4**	**207**	1.08	0.1
Rock Sample 5,5						
PVI	19.238±0.07	357±56	**21**±7	**362**±63	0.99±0.36	0.46±0.01
PPerseus	19.151±0.33	340±53	20±6	339±53	0.88±0.28	0.46±0.01
Perseus	19.08±0.36	413±56	228±252	10333±9777	60.34±66.62	0.05±0
PPBVI*	17.97	694	233	710	4.95	0.95
PBVI*	17.985	353	427	20616	72.01	0.49
HSVI	18.74	348	85	2309	10.39	0.26
Rock Sample 5,7						
PVI	22.945±0.41	358±88	**89**±34	359±89	1.28±0.64	0.29±0.01
Pperseus	22.937±0.70	408±77	118±37	407±77	1.61±0.59	0.29±0.01
Perseus	23.014±0.77	462±70	116±31	1002±195	5.18±1.9	0.02±0
PPBVI*	21.758	255	117	**254**	0.61	0.23
PBVI*	22.038	99	167	2620	3.05	0.15
HSVI	23.245	207	156	314	0.83	0.71

Table 4.2: Performance measurements. The algorithms that executed fewer backups and converged faster are bolded.



(a)



(b)

Figure 4.3: Convergence of various algorithms on the Rock Sample 5,5 problem (a) and the Tag Avoid problem (b). The X-axis shows the number of backups, the Y-axis shows the ADR.

Problem	$\lvert\bar{V}\rvert$	**Upper bound projections**	**#HV(b)**	$\lvert B\rvert$
Hallway	423	106132	1268	523
Hallway2	232	37200	434	171
Tag Avoid	1101	29316	1635	248
Rock Sample 4,4	344	6065	414	176
Rock Sample 5,5	801	101093	6385	1883
Rock Sample 5,7	3426	9532	628	268

Table 4.3: Upper bound measurements for HSVI.

PBVI and PPBVI failed in two cases (Rock Sample 5,5 and Rock Sample 5,7) to improve the reported ADR even when allowed more time to converge. These rows are marked with an asterisk.

4.2.7 Discussion

Our results clearly show how selecting the order by which backups are performed over a predefined set of points improves the convergence speed. When comparing PBVI to Prioritized PBVI and Perseus to Prioritized Perseus, we see that our heuristic selection of backups leads to considerable improvement in runtime. This is further demonstrated by the new PVI algorithm. In all these cases, there is an order of magnitude reduction in the number of backup operations when the next backup to perform is chosen in an informed manner. However, we also see that there is a penalty we pay for computing the Bellman error, so that the saving in backups does not fully manifest in execution time. Nevertheless, this investment is well worth it, as the overall performance improvement is clear. And although the ADR to which the different algorithms converge is not identical, the differences are minor, never exceeding 2%, making their ultimate ADR equivalent, for all practical purposes.

Examining Table **??**, we see that our PVI algorithm results in convergence time that is at least comparable, if not better, than existing point-based methods (PBVI, Perseus, and HSVI). The efficiency of its backup choices shows up nicely in Figure 4.3, where we see the steep improvement curve of PVI.

In many case, HSVI also executes a smaller number of backups than other algorithms. Indeed, one may consider HSVI's selection of belief space trajectories as a method for backup sequence computation and hence, as a prioritization metric. Nevertheless, in most cases our form of backup selection exhibits superior runtime to HSVI, even when the number of backups HSVI uses is smaller. This is due to the costly maintenance of the upper bound over the value function.

4.3 Forward Search Value Iteration[2]

Heuristic Search Value Iteration (HSVI; (Smith & Simmons, 2005)) is currently the most successful point-based algorithm for larger domains. HSVI maintains both a lower bound ($\underline{V}$) and an upper bound ($\bar{V}$) over the optimal value function. It traverses the belief space using a heuristic based on both bounds and performs backups over the observed belief points in reversed order. Unfortunately, using $\bar{V}$ slows down the algorithm considerably as updating $\bar{V}$ and computing upper bound projections ($\bar{V}(b)$) are computationally intensive. However, this heuristic pays off, as the overall performance of HSVI is better than other methods.

In this section we suggest a new method for belief point selection, and for ordering backups which does not use an upper bound. Rather, our algorithm uses a search heuristic based on the underlying MDP (the MDP that describes the environment). Using MDP information within POMDP algorithms is a well known technique. Specifically, it was used as an initial guess for the value function (starting with the Q_{MDP} method of (Kaelbling, Littman, & Cassandra, 1998)). Our novel algorithm — Forward Search Value Iteration (FSVI) — traverses together both the belief space and the underlying MDP space. Actions are selected based on the optimal policy for the MDP, and then applied to the belief space as well. As a result, the trajectory is directed towards the rewards of the domain.

4.3.1 FSVI

We propose a new algorithm, Forward Search Value Iteration (FSVI), using trial-based updates over the belief space of the POMDP. FSVI maintains a value function using α-vectors and updates it using point-based backups.

[2]Parts of this section appeared in (Shani et al., 2006)

The heuristic FSVI uses to traverse the belief space is based on the optimal solution to the underlying MDP. We assume that such a solution is given as input to FSVI in the form of a Q-function over MDP states. This assumption is reasonable, as a solution to the underlying MDP is always simpler to compute than a solution to the POMDP.

FSVI (Algorithm 17) simulates an interaction of the agent with the environment, maintaining both the POMDP belief state b and the underlying MDP state s — the true state of the environment the agent is at within the simulation. While at policy execution time the agent is unaware of s, in simulation we may use s to guide exploration through the environment.

Algorithm 17 FSVI

Function FSVI

Initialize V
while V has not converged **do**
  Sample s_0 from the b_0 distribution
  MDPExplore(b_0, s_0)

Function MDPExplore(b, s)

if s is not a goal state **then**
  $a^* \leftarrow \arg\max_a Q(s, a)$
  Sample s' from $tr(s, a^*, *)$
  Sample o from $O(a^*, s', *)$
  MDPExplore($\tau(b, a^*, o), s'$)
add($V, backup(b, V)$)

FSVI uses the MDP state to decide which action to apply next based on the optimal value function for the underlying MDP, thus providing a path in belief space from the initial belief state b_0 to the goal (or towards rewards). As we assume that the value function of the underlying MDP is optimal, this heuristic will lead the agent toward states where rewards can be obtained.

The trial is ended when the state of the underlying MDP is a goal state. When the MDP does not have a goal state, we can use other criteria such as reaching a predefined sum of rewards or number of actions. If the goal is unreachable from some states we may also add a maximal number of steps after which the trial ends.

FSVI (Algorithm 17) is apparently very simple. Its simplicity translates into increased speed and efficiency in generating belief points and updating the values associated with belief points. FSVI's method for selecting the next action is very fast, requiring to check only $O(|A|)$ values (MDP Q-values) as opposed to any action selection method based on the current belief state.

Other trial-based algorithms require much more time to select the next action. RTDP-BEL takes $O(|S|)$ operations for discretizing the belief state before it can select the action. HSVI needs $O(|A||O||S||\bar{V}|)$ operations, where $|\bar{V}|$ is the number of points in the upper bound, to compute the values of all the successors of the current belief state. As FSVI generates trajectories using forward projection, it is easy to determine good sequences of backups, simply going in reverse order. This ability is shared by HSVI, but not by other point-based algorithms such as Perseus (Spaan & Vlassis, 2005) and PBVI (Pineau et al., 2003).

Other algorithms, such as HSVI and RTDP-BEL, use a heuristic that is initialized based on the MDP Q-function and use some form of interpolation over these Q-values. These algorithms also improve the heuristic by updating the Q-values to fit the POMDP values. Such algorithms, therefore, work initially much like the Q_{MDP} heuristic which is known to perform badly for many POMDP problems and in many cases gets stuck in local optima. These algorithms can potentially need many updates to improve the heuristic to be able to reach rewards or goal states. FSVI, on the other hand, does not use a belief space heuristic based on the underlying

MDP, but uses the MDP directly. FSVI traversals lead very fast to rewards.

4.3.2 Value of Information

As noted earlier, a major drawback of MDP-based approaches is their inability to perform in information gathering tasks. FSVI is slightly different in that respect. FSVI uses point-based backups in which information gathering actions are also evaluated and considered. It is therefore able to perform single step information gathering actions such as the activation of a sensor. For example, in the RockSample domain (Smith & Simmons, 2005), the robot should activate a sensor to discover whether a rock is worthwhile. Indeed FSVI performs very well in the RockSample domain, executing such actions. However, when the information gathering requires a lengthy sequence of operations FSVI's heuristic will fail.

Consider for example the heaven-hell problem (Figure 4.4 (Bonet & Gefner, 1998)). On the topmost part of the maze there is a reward either in the left or the right cell. The cell marked 'M' contains a map telling the agent where the reward is hidden. The optimal policy is to walk to the map and then go to gather the reward. However, in the fully observable scenario, the agent is aware of the location of the reward without looking at the map. Therefore, the MDP policy will lead the agent directly towards the reward without traveling to the map. As the agent will not pass through the map cell, FSVI will not learn the policy that sends it to read the map and will fail to comptue an optimal policy.

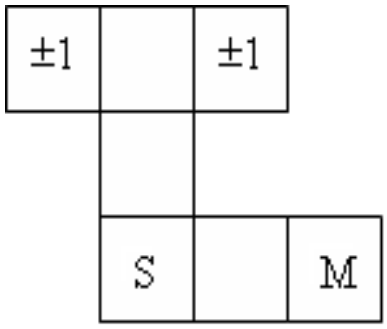


Figure 4.4: The Heaven-Hell problem. The heaven-hell cells are marked ±1. The agent starts at the cell marked S and the map is at the cell marked M.

To handle such domains, we must create a heuristic that computes trajectories that pass through the states that provide important observations. This process can be split into 3 stages:

1. Discover the need for information.
2. Find states where the required information can be gathered.
3. Devise trajectories that pass through these states.

In the following subsections we shall propose a solution to all the required stages.

Discovering Information Insufficiency

We shall focus here on factored POMDP representations, where states are defined by an assignment to state variables $X_1, ..., X_n$. The natural way to formalize the heaven-hell problem uses a factored form with two state variables — X_1 holds the agent location in the environment and X_2 holds the location of the reward. We will attempt to discover state variables that have an effect on the reward.

We say that two states s and s' differ only on variable X if the value of X in s is different than the value of X in s'. For every other state variable $X' \neq X$, X' has the same value in s and in s'.

A state variable X influences a reward if there exist two states s and s' such that:

1. s and s' differ only on X.
2. There exists a reachable belief state b such that $|b(s)Q(s, \pi_{MDP}(s)) - b(s')Q(s', \pi_{MDP}(s))|$ is greater than some constant k, meaning that the expected value of executing $\pi_{MDP}(s)$ in b has a high variance.

In practice, we are interested in state variables that influence the reward only if the belief state b above can be found on some trajectory FSVI traverses. During FSVI traversals, we need to check each belief state to find whether it contains such two states. If we discover such a belief state, then we are in an environment where additional information regarding X can increase the expected reward.

Finding Informative States

We now need to find states where information about X can increase the reward. Information about X can be obtained by some observation o that is influenced by the value of variable X. We thus look for states s, s' that:

1. s and s' differ only on X.
2. For some action a and some observation o, $|O(a, s, o) - O(a, s', o)|$ is greater than some constant δ.

Even though a simple implementation would require $O(|S|^2)$ operations for finding such states, in a factored representation where system dynamics are specified as a dynamic Baeysian network (DBN), the DBN supplies such information immediately.

Constructing Information Aware Sequences

After finding states where we can gather information about X, we now need to direct FSVI towards these states. There is no need to make FSVI attempt to execute information gathering actions, as these will be considered by the point-based backup.

We direct FSVI towards these states by altering the reward function of the underlying MDP. We do so by adding a large reward for each state where information can be gathered over X. This reward can be obtained only once in every trial. This is done by adding a state variable for each such state, specifying whether the reward has been collected. The reward that we add must be large enough so that the optimal policy will attempt to collect it before it collects the 'true' rewards of the system.

We now compute the optimal policy for our revised MDP. This policy will gather first the artificial rewards, due to the discount factor. After collecting these rewards, the policy will continue to the 'true' rewards. We shall now execute FSVI using the policy for the revised MDP rather than the policy for the original underlying MDP.

4.3.3 Empirical Evaluations

4.3.4 Evaluation Metrics

We evaluate the following criteria:

Value function evaluation — Average discounted reward (ADR): $\frac{\sum_{i=0}^{\#trials}\sum_{j=0}^{\#steps}\gamma^j r_j}{\#trials}$. ADR is filtered using a first order filter to reduce the noise in ADR.

Execution time — we report the CPU time, but as this is an implementation-specific measurement, we also report the number of basic operations such as backup, τ function, dot product ($\alpha \cdot b$), and $g^{\alpha}_{a,o}$ (Equation 4.6) computations .

Memory — we show the size of the computed value function and the number of maintained belief points.

Experimental Setup

The most obvious candidate for comparison to FSVI is HSVI (Smith & Simmons, 2005) as both algorithms are trial based and conduct a form of heuristic forward search. Also, HSVI has so far shown the best performance on large scale domains. Algorithms such as Perseus (Spaan & Vlassis, 2005), PBVI (Pineau et al., 2003), and PVI, that constantly maintain a large set of belief points, have trouble scaling up to larger domains. Trial-based algorithms that keep only a single trajectory scale up better.

Comparison is done on a number of benchmarks from the point-based literature: Hallway, Hallway2 (Kaelbling et al., 1998), TagAvoid (Pineau et al., 2003), RockSample (Smith & Simmons, 2005), and Network Ring (Poupart & Boutilier, 2002). These include all the scalability domains on which HSVI was tested in (Smith & Simmons, 2005), except for Rock Sample 10,10 which could not be loaded on our system due to insufficient memory. Table 4.4 contains the problem measurements for the benchmarks including the size of the state space, action space and observation space, the number of trials used to evaluate the solutions, and the error in measuring the ADR for each problem.

The Rock Sample domain provides an opportunity for testing the scalability of different algorithms. However, these problems are somewhat limited: they assume deterministic state transitions as well as full observability for the robot location, making the problems easier to solve. To overcome these limitations we added a new domain — Noisy Rock Sample, in which agent movements are stochastic and it receives noisy observations as to its current location.

Problem	$\|S\|$	$\|A\|$	$\|O\|$	#trials	ADR Error
Hallway	61	5	21	10,000	±0.0015
Hallway2	93	5	17	10,000	±0.004
Tag Avoid	870	5	30	10,000	±0.045
Rock Sample 4,4	257	9	2	10,000	±0.075
Rock Sample 5,5	801	10	10	10,000	±0.3
Noisy RS 5,5	801	10	27	10,000	±0.3
Rock Sample 5,7	3,201	12	2	10,000	±0.25
Rock Sample 7,8	12,545	13	2	1,000	±0.25
Rock Sample 8,8	16,385	13	2	1,000	±0.25
Network Ring 8	256	17	2	2,000	±1.1
Network Ring 10	1024	21	2	2,000	±0.98

Table 4.4: Benchmark problem parameters

We implemented in Java a standard framework that incorporated all the basic operators used by all algorithms such as vector dot products, backup operations, and τ function. All experiments were executed on identical machines: x86 64-bit machines, dual-proc, 2.6Ghz CPU, 4Gb memory, 2Mb cache, running linux and JRE 1.5.

Method	**ADR**	$\lvert V\rvert$	**Time** (secs)	**#Backups**	$\#g^{\alpha}_{a,o}$ x 10^6	$\lvert B\rvert$ x 10^4	$\#\tau$ x 10^3	$\#\ \alpha \cdot b$ x 10^6	$\lvert\bar{V}\rvert$ 10^3	$\bar{V}(b)$ 10^3	$\#H\bar{V}$
Hallway				187ms	0.13μs		0.37μs	33ns		0.5μs	65ms
HSVI	0.516	182	314	634	5.85	3.4	34.52	6.67	0.42	106.1	1268
FSVI	0.517	233	50	655	7.71	0.05	0.51	7.78			
	±0.0024	±71	±15	±100	±2.46	±0.01	±0.08	±2.49			
Hallway2				222ms	0.17μs		1μs	36ns		1μs	126ms
HSVI	0.341	172	99	217	1.56	2.11	11.07	1.81	0.23	37.2	434
FSVI	0.345	296	49	355	4.28	0.03	0.3	4.33			
	±0.0027	±22	±8	±33	±0.73	±0	±0.03	±0.74			
Tag Avoid				311ms	0.92μs		0.56μs	8.3ns		0.6μs	24.1ms
HSVI	-6.418	100	52	304	0.5	0.29	1.74	0.53	1.1	29.3	1635
FSVI	-6.612	174	45	182	0.37	0.01	0.14	0.39			
	±0.15	±25	±8	±27	±0.1	±0	±0.02	±0.11			
Rock Sample 4,4				92ms	0.2μs		0.46μs	0.031μs		0.2μs	4.67ms
HSVI	18.036	123	4	207	1.08	0.1	1.17	1.09	0.34	6.06	414
FSVI	18.029	84	1	204	0.24	0	0.04	0.25			
	±0.024	±76	±1	±122	±0.41	±0	±0.01	±0.42			
Rock Sample 5,5				114ms	1.1μs		2.5μs	0.033μs		0.64μs	24.1ms
HSVI	18.74	348	85	2309	10.39	0.26	2.34	10.5	0.8	101.1	1883
FSVI	19.206	272.5	11.1	626.2	1.47	0.02	0.1	1.49			
	±0.063	±75	±5	±247	±0.88	±0	±0.02	±0.89			
Noisy Rock Sample 5,5				314ms	0.86μs		1.9μs	0.035μs		0.69μs	44.5ms
HSVI	10.318	2639	1586	3528	129.11	2.01	23.05	132.4	0.88	264.9	9294
FSVI	10.382	924	210	1153	11.93	0.48	0.53	12.09			
	±0.069	±170	±52	±224	±4.79	±0.12	±0.14	±4.8			
Rock Sample 5,7				143ms	3.7μs		26.6μs	0.11μs		3.0μs	256.6ms
HSVI	22.67	274	205	350	0.65	1.0	4.3	0.99	3.44	14.09	702
FSVI	22.82	306.9	34.3	500	39684	3722	0.4	2.1			
	±0.63	±91.5	±13.6	±400.1	±0.02	±0.014	±0.001	±2.9			
Rock Sample 7,8				567ms	20μs		0.25ms	1.1μs		15μs	3.4sec
HSVI	20.029	533	3045	2089	14.51	1.5	4.1	14.66	3.42	9.53	628
FSVI	20.369	343.1	239	512.1	2.389	0.049	0.024	2.457			
	±0.265	±146.6	±78.7	±284.8	±2.32	±0.059	±0.013	±2.357			
Rock Sample 8,8				570ms	25μs		0.35ms	2.6μs		18μs	2.7sec
HSVI	19.474	762	13917	1317	10.83	4.01	58.09	11.43	17.3	58.38	2638
FSVI	19.646	261.4	783	367.8	0.042	0.816	0.176	1.183			
	±0.337	±76.9	±295	±125.1	±0.02	±0.013	±0.06	±0.661			
Network Ring 8				164ms	1μs		2.6ms	4.5ns		11μs	31.8ms
HSVI	42.27	80	19	153	0.004	0.25	8.44	0.31	0.39	8.46	307
FSVI	42.39	40.5	6.75	252	0.004	0.022	0.24	0.19			
	±0.18	±16	±6.1	±146	±0.002	±0.012	±0.14	±0.19			
Network Ring 10				553ms	10.6μs		13.3ms	23.3ns		99.4μs	369ms
HSVI	51.43	69	141	103	0.0036	0.29	6.9	0.144	1.11	6.9	206
FSVI	51.44	33.25	47	267.7	0.002	0.025	0.255	0.19			
	±0.03	±6.14	±14.3	±85.78	±0.0004	±0.008	±0.081	±0.086			

Table 4.5: Performance measurements. Model rows show rough estimates of basic operations execution time.

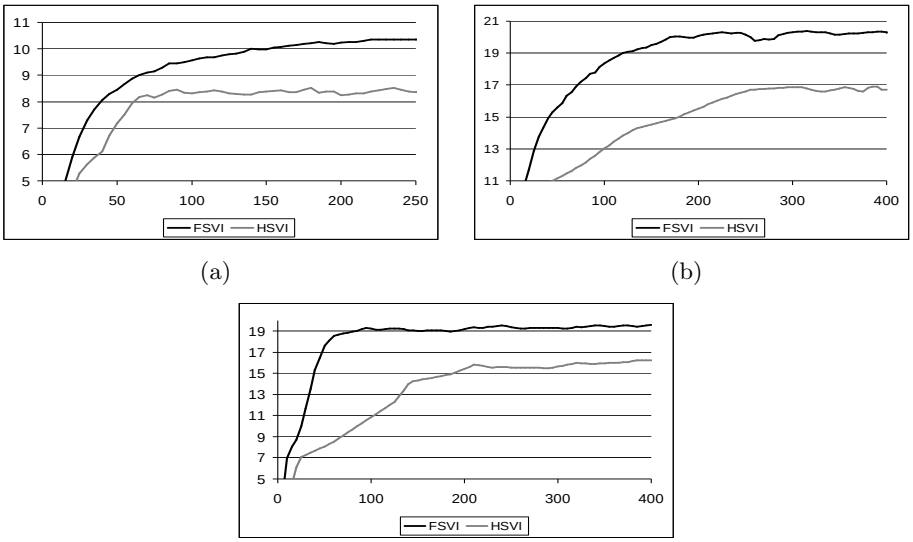


Figure 4.5: Convergence on the Noisy Rock Sample 5,5 problem (a), the Rock Sample 7,8 problem (b), and the Rock Sample 8,8 problem (c). The X-axis shows CPU time and the Y-axis shows ADR.

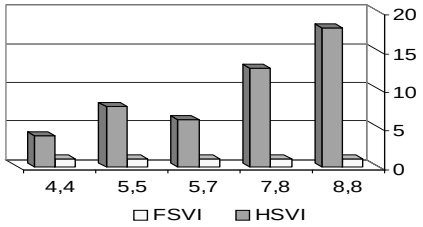


Figure 4.6: Normalized comparison of CPU time for the Rock Sample problems.

We focus our attention on convergence speed of the value function to previously reported ADR. We executed HSVI and FSVI, interrupting them from time to time to compute the efficiency of the current value function using ADR. Once the filtered ADR has reached the same level as reported in past publications, execution was stopped. The reported ADR was then measured over additional trials (number of trials and error in measurement is reported in Table 4.4).

Results

Table 4.5 presents our experimental results. For each problem and method we report:

1. Resulting ADR.
2. Size of the final value function ($|V|$).
3. CPU time until convergence.
4. Backup count.
5. $g^{\alpha}_{a,o}$ operations count.

6. Number of computed belief states.
7. τ function computations count.
8. Dot product operations count.
9. Number of points in the upper bound ($|\bar{V}|$).
10. Upper bound projection computations count
11. Upper bound value updates count ($H\bar{V}(b)$).

The last 3 items refer only to HSVI, as FSVI does not maintain an upper bound. The reported numbers do not include the repeated expensive computation of the ADR, or the MDP value function computation (identical for both algorithms).

To illustrate the convergence rate of each algorithm, we also plotted the ADR as a function of CPU time in Figure 4.5. These graphs contain data collected over separate executions with fewer trials, so Table 2 is more accurate.

We also experimented with the heaven-hell domain to test our approach for solving problems that require lengthy information sequences. FSVI easily found the optimal policy for this domain, after executing 26 backups. On the same problem, HSVI required 442 backups.

4.3.5 Discussion

Our results clearly indicate that FSVI converges faster than HSVI and scales up better. The convergence rate shown in Figure 4.5 is always faster, the time required to reach optimal ADR is also always faster, and this difference become more pronounced as problem size increases. A more focused comparison of the relative ability of the two solvers to scale up is presented in Figure 4.6. Overall, it appears that HSVI is slowed down considerably by its upper bound computation and its updates. While it is possible that HSVI's heuristic leads to a more informed selection of belief points, FSVI is able to handle more belief points, and faster. In fact, in the Rock Sample domains, we can see that FSVI is able to converge with fewer belief points and even slightly improved ADR. Thus, on these domains, the heuristic guidance in point selection offered by the MDP policy is superior to the bounds-based choice of HSVI. Indeed, we can also see that the number of backups FSVI executes is in most cases less than HSVI, hinting that the chosen trajectories (and hence backups) are also better than those selected by HSVI.

Observe that the upper bound of HSVI allows the definition of additional properties such as a stopping criterion based on the gap between bounds over the initial belief point b_0, and an upper bound over the number of trials required to close that gap. In practice, however, HSVI never manages to reduce this gap considerably (as also noted in (Smith & Simmons, 2005)) and reaches maximal performance when the gap is still very large. This is to be expected, as the convergence guarantees for HSVI rely on the eventual exploration of the entire And-Or search graph for the domain, until an appropriate depth that grows as γ approaches 1 is located.

Our modification to the FSVI search heuristic, designed to visit states where information concerning the location of rewards can be gathered, has shown very good results on the heaven-hell problem. Fully solving the lack of information problem using finite size MDP-based approaches does not seem like a possible task. Yet, our approach provides an important step forward towards making FSVI applicable for all POMDP benchmarks.

4.4 Efficient ADD Operations for Point-Based Algorithms

In many cases, it is natural to describe the state of the environment within a POMDP via a set of *state variables*, and the effects of actions in terms of their effects on these variables. Dynamic Bayesian Networks (DBNs) with conditional probability tables (CPTs) in the form of decision-trees or graphs are often used to represent these effects compactly. This representation is often referred to as a *factored* representation. The state space of such models is exponential in the number of variables, and quickly grows outside the reach of methods that operate on an explicit, flat representation, including point-based methods. To address this problem, researchers suggested the use of Algebraic Decision Diagram (ADDs) (Boutilier & Poole, 1996; Hansen & Feng, 2000; Poupart, 2002). ADDs represent functions $f : \{T/F\}^n \rightarrow R$ and can therefore compactly represent the environment dynamics. ADDs support fast function sum and product operations. Consequently, all basic POMDP operations, such as the computation of the next belief state, can be implemented efficiently on ADDs. When each action affects a relatively small number of variables, this representation can be very compact.

ADD-based algorithms for MDPs/POMPDs are not new. In the context of fully observable Markov Decision Processes (MDPs), ADD-based algorithms scale-up better than algorithms that operate on flat representations (St-Aubin, Hoey, & Boutilier, 2000). Adapting these methods to POMDPs, Feng and Hansen explain how α-vector computation can be implemented using ADDs and how α-vectors can be compacted (Hansen & Feng, 2000). Poupart and Hoey et al. (Poupart, 2002; Hoey, von Bertoldi, Poupart, & Mihailidis, 2007) used the methods of Hansen and Feng in an implementation of the Perseus point-based algorithm (Spaan & Vlassis, 2005). However, as we show below, these implementations do not scale up well to larger problems.

In this section we show how basic operations used in the context of point-based algorithms can be implemented efficiently using ADDs and, consequently, how ADDs can be used in point-based algorithms. We explore well known ADD optimizations such as state variable ordering and merging of similar values to reduce the size of the ADD. We also report a number of new optimizations that reduce the time used to compute inner products and backup operations.

In addition, we take a special look at domains that possess effect locality, i.e., the property that each action affects only a small number of variables. Many standard test domains exhibit this property, and other authors attempted to utilize it (Guestrin, Koller, & Parr, 2001). This paper explains how backup and τ operations used in point-based methods can be efficiently implemented for such domains. We show experimentally that in domains exhibiting reasonable effect locality, our ADD-based point-based algorithms are several orders of magnitude faster than the best existing point-based algorithms, considerably pushing the boundary of existing POMDP solution algorithms.

(Boutilier & Poole, 1996) suggested representing factored POMDPs using decision trees, where nodes are labeled by state variables and edges are labeled by possible values for the state variables (T/F). The decision tree leaves contain real numbers, representing probabilities or values.

4.4.1 Algebraic Decision Diagrams (ADDs)

An Algebraic Decision Diagram (R.I. Bahar et al., 1993) is an extension of Binary Decision Diagrams (Bryant, 1986) that can be used to compactly represent decision trees. A decision tree can have many identical nodes, and an ADD unifies these nodes, resulting in a rooted DAG rather than a tree. The ADD representation becomes compact as the structure in problem definition grows. A decision tree compacted into an ADD is shown in Figure 4.7.

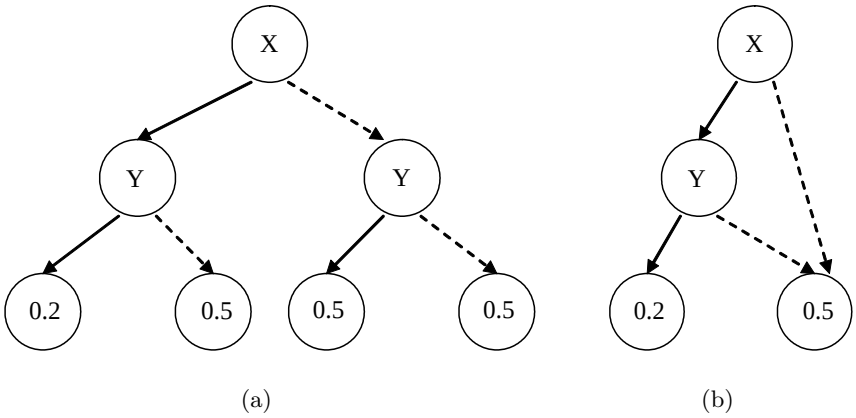


Figure 4.7: A decision tree and an ADD of the same function.

As ADDs define functions $f : \{T/F\}^n \rightarrow R$, function *product* (denoted $\otimes$), function *sum* (denoted $\oplus$), and *scalar product* or inner product (denoted $\odot$) can be defined.

Given a variable ordering, an ADD has a unique minimal representation. The *reduce* operation takes an ADD and reduces it to its minimal form.

Another operation specialized for ADDs is the *existential abstraction*, also know as variable elimination (denoted $\sum_{X_i}$). This operation eliminates a given variable X_i from the ADD by replacing each occurrence of X_i by the sum of its children.

The *translate* operation (denoted $A(Y \rightarrow Z)$), takes an ADD and replaces each occurrence of variable Y by variable Z. Z must not appear in the ADD prior to the translation.

The value an ADD assigns to some specific variable assignment $< x_1, ..., x_n >$ is denoted $A(x_1, ..., x_n)$.

Most ADD operations we use were implemented following the algorithms outlined by (Bryant, 1986) with a number of modifications described in the following sections. We did not use any existing ADD package.

4.4.2 Point-Based Value Iteration with ADDs

An ADD-based point-based algorithm uses ADDs to represent belief states and α-vectors, and must provide an ADD-based implementation of the basic operations used: belief state backup (Equations 4.6 and 4.7), belief state update (Equation 2.7), and inner product of a belief state and an α-vector (Equation 2.22). In this section we explain how to implement these operations (Boutilier & Poole, 1996; Hansen & Feng, 2000; Poupart, 2002).

ADD-based Operations

To compute both τ and $g^{\alpha}_{a,o}$ we create P^o_a — the *Complete Action Diagram* (Boutilier & Poole, 1996; Hansen & Feng, 2000) for each action a and observation o.

$$P^o_a(x_1, ..., x_n, x'_1, ..., x'_n) = tr(< x_1, ..., x_n >, a, < x'_1, ..., x'_n >)O(a, < x'_1, ..., x'_n >, o) \quad (4.14)$$

We also use an ADD format to represent the immediate reward function:

$$R_a(x_1, ..., x_n) = R(< x_1, ..., x_n >, a) \quad (4.15)$$

The set of all P^o_a and R_a ADDs are a sufficient description of the system dynamics.

Belief update: Using the complete action diagram we replace Equation 2.7, computing the next belief state with:

$$b' = (\sum_{X_1,\ldots,X_n} P_a^o \otimes b)(X' \rightarrow X) \tag{4.16}$$

where the $\otimes$ operation is the ADD product and the sum is an existential abstraction over the pre-action state variables. We now need to do a scalar product over the values in the resulting ADD by $1/p(o|a,b)$. A straight forward way to compute $pr(o|a,b)$ is to eliminate all variables in the resulting ADD, but as we show in the next section $p(o|a,b)$ can be easily computed.

After the existential abstraction operation has removed all pre-action variables, the remaining ADD is over X' — the post-action variables. We now translate the variables back into pre-action variables, replacing each occurrence of any X_i' with X_i.

Backup: To replace Equation 4.6 for computing $g_{a,\alpha}^o$ with an ADD implementation we use a similar approach:

$$\alpha' = \sum_{X_1',\ldots,X_n'} P_a^o \otimes \alpha(X \rightarrow X') \tag{4.17}$$

The translation here of the former α-vector (ADD) is needed because Equation 4.6 requires the value of the post-action state s'. All other operations used for backup such as summing and scalar products are done over the ADDs.

Inner product: Inner product operations provide a challenge. A simple implementation that follows the algorithm structure of (Bryant, 1986) will compete poorly against the flat method. It is easy to implement inner products very efficiently using sparse (non-zero entries only) vector representations of a belief state and α-vector. Indeed, our experiments indicate that such implementation performs poorly. We explain in the next section how to efficiently implement inner products.

Compressing ADDs

Feng and Hansen show how the structure in an ADD can be exploited to create approximations that influence the size of the ADD. We can reduce the number of leaves, and hence the number of nodes, by grouping together values that are ϵ far, thus creating smaller ADDs with little loss of accuracy. This operation can be easily done for ADDs representing α-vectors. To ensure that the α-vectors remain a lower bound over the value function, we always replace a value v with a smaller value in the range $[v-\epsilon, v]$.

The same process is applicable for belief states. However, for belief states where values can be substantially smaller than α-vector values, we use a much smaller ϵ. Also, we make sure that a non-zero value is never transformed into a zero value in the process.

Symbolic Perseus

A previous attempt to implement a point-based algorithm using ADDs was done by Poupart (Poupart, 2002) and Hoey et al. (Hoey et al., 2007). Poupart used ADDs for an implementation of the Perseus point-based algorithm (Spaan & Vlassis, 2005), calling the resulting algorithm Symbolic Perseus. Symbolic Perseus uses a bounded value function, i.e., it does not allow its value function to grow beyond a pre-defined number of vectors, in order to reduce the time required for a backup operation.

Symbolic Perseus uses a belief state approximation method designed to reduce the size of the belief state (Boyen & Koller, 1998). Hopefully, reducing the belief state size will also result

in decreased computation time of belief state related operations. An approximate belief state is composed of distinct sets of state variables, called components. The belief is maintained independently over the sets. When computing a new belief state given $\tilde{b}$ — the approximate belief state, an action a, and an observation o, we first compute $b' = \tau(\tilde{b}, a, o)$ the exact next belief state. We then project b' into the space of partitioned beliefs creating $\tilde{b'}$ — the next approximate belief state.

For implementing this approach with ADDs, we must maintain, instead of the complete action observation diagram, a set of ADD $T_{a,i}$ specifying the transition probabilities for the state variables in component i at the next time step given all state variables at the current time step and an action a. We also require $O_{a,o}$ — an ADD specifying the probabilities of observing o after executing action a.

In ADD operations, given a set $\tilde{B} = \{\tilde{B}_i\}$ of ADDs, specifying the probabilities of the independent components, we first compute the exact next belief state ADD:

$$B' = O_{a,o} \prod_i \tilde{B}_i T_{a,i} \tag{4.18}$$

The resulting B' requires normalization. Then, for each new belief state component:

$$\tilde{B}'_i = (\sum_{X \cup X' / Y'_i} B')(X' \rightarrow X) \tag{4.19}$$

that is — for each component in the new belief state we use existential abstraction to remove all other variables.

Another operation that must be redefined is the inner product of a belief state and an α-vector:

$$\alpha \cdot \tilde{B} = \alpha \odot \prod_i \tilde{B}_i \tag{4.20}$$

However, Symbolic Perseus implements the inner product operation more efficiently. It is possible to replace the above implementation with a traversal of the α-vector ADD. When at a vertex labeled by variable X_i, we can compute the local probability of each value of X_i given the relevant belief ADD for X_i. We can hence multiply the inner product value received from the children of the vertex with the probability of that child. This implementation is much faster and was therefore used in our experiments below.

(Poupart, 2002) experimented with the Network Administrator problem with increasing size, concluding that ADD operations do not scale up well. Hoey and Poupart (Hoey et al., 2007) show Symbolic Perseus solves a specific factored POMDP of a large size, but with a very special structure.

4.4.3 Scaling ADD Operations

In flat POMDPs, belief states and α-vectors are efficiently implemented as vectors containing only the non-zero entries. An ADD representation can be beneficial when belief states and/or α-vectors contain much structure. In the worst case, the ADD reduces to a complete decision tree, and the quantity of storage space is twice that of a flat vector.

Unfortunately, the straight-forward replacement of vectors with ADDs discussed in the previous section scales poorly. In all the domains we experimented with, an efficient flat implementation is faster. We describe below a set of improvements that leads to substantially better performance. All of the improvements we suggest do not introduce more approximations to belief states or α-vectors.

Our improvements address both the size of the ADD as well as the speed of ADD-based operations. We consider:

Variable ordering: Decreasing the size of P_a^o using a different ordering of the state variables in the ADD.

Inner product: Augmenting ADD structure to provide rapid inner product operations.

Relevant variables: Removing some state variables from P_a^o, resulting in a substantial decrease in ADD size and increase in computation efficiency.

Backup operation: Efficiently implementing point-based backups using τ function operations. While this improvement is not directly related to the ADD structure, it is best manifested when using ADDs.

Variable Ordering

As ADD operations (e.g., product and sum) are linear in ADD size, a decrease in size results in a substantial speedup.

(Boutilier & Poole, 1996) and (Hansen & Feng, 2000) create P_a^o (Equation 4.14) by joining the original ADDs that specify the single variable transition probabilities. This method is very fast, avoiding explicit enumeration of state variable values, but orders all post-action variables X' before the corresponding pre-action variables X. Such an ordering is undesirable in many cases. In a maze navigation problem, for example, a robot is able to transition from the current cell only to adjacent cells. Specifying such a behavior when all X‘ precedes X forces us to create paths that first contain all X' variables and only afterwards when a single X_i has an invalid assignment ($x_i \neq x_i'$) a 0 valued leaf is reached, denoting the illegal transition. This results in a complete tree until depth $|X|$ which is exponential in the number of state variables.

It is better to place every X_i immediately after (or before) X_i'. This was also noted in the context of MDPs by (St-Aubin et al., 2000). Unfortunately, we can no longer use the method described by Boutiler and Poole for creating P_a^o. Instead, we fully specify each variable assignment path $< x_1', x_1, ..., x_n', x_n >$ and add the path and its value to the ADD, resulting in a complete decision tree. We can then reduce the ADD to its minimal form. When most transitions have zero probability, we improve this process by adding only paths ending in non-zero values, and replacing all the null children of nodes in the tree with the value 0. Path based construction supports any variable ordering.

While computing P_a^o and R_a using an enumeration over all the possible values of the state variables is time consuming, it is done once as a pre-computation step. The resulting complete action diagrams and reward diagrams are used as the complete model description. There is no longer any need to maintain other structures for the transition, reward, or observation functions.

Efficient Inner Product Operations

The inner product operation between a belief state and an α-vector is executed many times by point-based algorithms. Within the augmented backup process described below and when attempting to find the best α-vector for a belief state during policy execution, many inner products are used.

A possible implementation of inner-products can handle the ADDs as if they were fully specified decision trees, traversing the two trees together, multiplying leaf scalars, and summing the results. Most ADD operations such as product and sum are implemented likewise, and can be augmented by caching paths that were already traversed. Another, even more costly alternative, is to first compute the product of the two ADDs and then eliminate all variables through existential abstraction, resulting in a single leaf containing the sum of the scalars in the ADD.

As the result of an inner product is not an ADD but a scalar, we suggest a more efficient solution that does not traverse the entire ADD structure. We first add to each ADD vertex a scalar, specifying the sum of the subtrees below it. As in an ADD, an efficient representation may not specify some variables, we need to be careful when computing the sum such that missing variables are still taken into consideration. For example, in the ADD in Figure 4.7, the value sum of the Y variable is $0.2 + 0.5 = 0.7$, but when we compute the value sum of the X variable it is $0.7 + 0.5 \times 2 = 1.7$ because the right subtree of X contains two instances of the 0.5 value (as we can see in the fully specified decision tree) represented by a single node. Computing the sum of values is done within the reduce operation of the ADD and does not require additional runtime.

Using the computed sum at each node, we can avoid traversing all the nodes of the ADDs. Algorithm 18 outlines this operation[3].

Algorithm 18 InnerProduct(v_1,v_2)

1: **if** v_1 is a leaf **then**
2: $c \leftarrow$ number of decision tree leaves compressed in v_1
3: **return** $value(v_1) * c + valueSum(v_2)$
4: **if** v_2 is a leaf **then**
5: $c \leftarrow$ number of decision tree leaves compressed in v_2
6: **return** $value(v_2) * c + valueSum(v_1)$
7: **return** $InnerProduct(leftChild(v_1), leftChild(v2))$+$InnerProduct(rightChild(v_1), rightChild(v2))$

As we can see, the recursion terminates whenever the traversal over one of the ADDs reaches a leaf. Hence, computation time depends on minimal paths rather than on maximal paths, as in most ADD operations. For example, when computing the inner products of the two ADDs in Figure 4.8, only the grey nodes are visited. An extensive literature search we conducted did not find a similar algorithm.

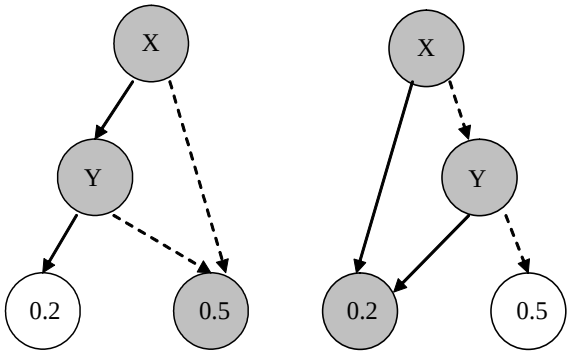


Figure 4.8: Two ADDs with different structures. During an inner product operation, only nodes colored in grey are visited.

Adding the sum of its children to each node has another helpful aspect. When computing a belief state update (Equations 2.7 and 4.18) we need to normalize the resulting belief state. The normalizing value — $pr(o|a, b)$ — is the sum of values of the ADD. While it is possible to receive this value by an existential abstraction over all the variables of the ADD, the value sum of the root is the sum of all values of the ADD.

[3]Some details were omitted to allow for more readable pseudo-code.

Relevant Variables

The size of P_a^o – the complete action diagram – influences the computation speed of all operations. Variable ordering provides a substantial reduction but a further reduction can be achieved by ignoring some state variables.

In many problems each action affects only a small subset of the state variables of the domain. For example, in the RockSample domain, when a robot takes a *move* action, the state of the rocks (Good/Bad) does not change. We define X_R^a the set of state variables affected by action a, and $X_{\neg R}^a$, the set of state variables unchanged by a:

$$X_R^a = \{X_i : pr(X_i' \neq x | X_i = x, a) > 0\} \tag{4.21}$$

$$X_{\neg R}^a = \{X_i : pr(X_i' = x | X_i = x, a) = 1\} \tag{4.22}$$

$$pr(o|X', a) = pr(o|X_R'^a, a) \tag{4.23}$$

The last equation specifies that the observation received after action a depends only on the state variables in X_R^a.

Now, we can simplify Equation 4.16:

$$\sum_{X_1,\ldots,X_n} P_a^o \otimes b = \sum_{X_1,\ldots,X_n} P_{R_a}^o \otimes P_{\neg R_a} \otimes b \tag{4.24}$$

where $P_{R_a}^o$ is an ADD restricted to the relevant variables and $P_{\neg R_a}$ is restricted to the irrelevant variables. $P_{\neg R_a}$ contains only variables that are unaffected by action a and also do not affect the probability of receiving an observation afterwards, and hence does not depend on the observation o.

Computing this equation using two ADD product operations is inefficient, as pre-computing the complete action diagram is done once where Equation 4.24 is executed over each τ operation. However, $P_{\neg R_a} \otimes b$ can be computed without executing ADD product. Recall that for each $X_i \in X_{\neg R}^a$, $p(X_i' = x | X_i = x, a) = 1$, i.e., the variable value does not change with action execution. Hence, the ADD product reduces to a simple translation operation – replacing each $X_i \in X_{\neg R}^a$ in b by X_i':

$$\sum_{X_1,\ldots,X_n} P_a^o \otimes b = \sum_{X_1,\ldots,X_n} P_{R_a}^o \otimes b(X_{\neg R}^a \rightarrow X_{\neg R}'^a) \tag{4.25}$$

In fact, $P_{\neg R_a}$ is unneeded and uncomputed by our algorithm.

An equivalent change is done to Equation 4.17, required for backup computations.

Instead of the complete action diagrams, we hence maintain partial action diagrams over the relevant variables only. This results in smaller, and in some cases, much smaller ADDs and in rapid computations of backups and τ operations.

Efficient Backup Operations

Measuring the CPU time it takes to execute $g_{a,o}^\alpha$ operations and comparing it to τ operations it is apparent that τ operations are up to 100 times faster. We believe that this is because α-vectors rapidly become less structured, causing their ADDs to grow in size while, in the examples we use, belief states are highly structured and remain so even after executing many actions from the initial belief point b_0.

Therefore, computing the value of a belief state in the next value function by Equation 4.1, which is done by computing many belief updates, is faster than computing the backup operation (Equation 4.7) that computes many $g_{a,o}^\alpha$.

Some of the computed $g^{\alpha}_{a,o}$ will later be used to construct the new α-vector in Equation 4.6. However, most of the $g^{\alpha}_{a,o}$ computed by the backup operation are dominated by others and will not participate in the newly created α-vector. If we knew the action a and the vectors α_i that maximize Equations 4.6 and 4.7, we could avoid computing dominated $g^{\alpha}_{a,o}$. Indeed, the action and α-vectors that maximize these equations are exactly the same as those that maximize Equation 4.1, which suggests a more efficient implementation for the backup process.

Algorithm 19 uses Equation 4.1 to compute which $g^{\alpha}_{a,o}$ are combined to yield the resulting α-vector and only then computes and combines them, thus minimizing the needed number of $g^{\alpha}_{a,o}$ computations.

Algorithm 19 Backup(b,V)

```
1: a* ← nil, v* ← −inf
2: for each o ∈ ω do α*_o ← nil
3: for each a ∈ A do
4:     Q_a(b) ← 0
5:     for each o ∈ ω do
6:         b' ← τ(b, a, o)
7:         α_o ← arg max_{α∈V} α · b'
8:         Q_a(b) ← Q_a(b) + α_o · b'
9:     if Q_a(b) > v* then
10:        a* ← a, v* ← Q_a(b)
11:        for each o ∈ ω do α*_o ← α_o
12: return r_{a*} + γ Σ_o g^{α*_o}_{a,o}
```

As all $g^{o}_{a,\alpha}$ computations are replaced with τ Computations, the complexity of the backup process does not change. Nevertheless, the actual CPU time is reduced considerably.

Poupart et al. (Poupart, Vlassis, Hoey, & Regan, 2006) were motivated by the same intuition when presenting their backup operation for a special structure POMDP, where states can be used instead of observations. They did not present any experimental results showing how their modified backup process compares to the standard backup process (Pineau et al., 2003). They also did not show how their method generalizes to regular point-based algorithms for POMDPs.

4.4.4 Experimental Results

We tested our approach on a number of domains with increasing size. Our goal is to evaluate the ADD-based approach and determine whether it allows us to scale better to larger domains. We also evaluate the influence of effect locality on ADD operations.

In the tables below we show a detailed comparison of the running time of flat and ADD-based implementations of the basic operations used by point-based algorithms: τ function, $g^{\alpha}_{a,o}$ operations, inner products, and the full backup process. We also provide total running time. Our tests use the FSVI algorithm which is both very simple and highly efficient. However, our ADD operations can be used by any point-based algorithm.

Our flat representation is also as efficient as possible. All belief points and α-vectors maintain only non-zero entries and all iterations are done only over these entries. All operations that iterate over the set of states $S_{a,s} = \{s' : tr(s, a, s') \neq 0\}$ for some state s are implemented in $O(|S_{a,s}|)$, avoiding zero transitions.

Benchmark Domains

RockSample – In the RockSample domain (Smith & Simmons, 2005) a rover is scanning an

Rocks	6	7	8	9	10	11	12	13	14	15
$\|S\|$	2^{12}	$\|S\|$	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
Average time for $g^{\alpha}_{a,o}$ computation in milliseconds (10^{-3})										
Flat	4993	19058	77904	310354	×	×	×	×	×	×
Factored	4	7	23	37	74	360	432	644	1321	1346
Average time for τ computation in microseconds (10^{-6})										
Flat	13952	46305	281953	946351	1455786	×	×	×	×	×
Factored	315	349	375	463	585	642	701	1007	1115	1342
Independent	1980	2317	2676	3073	3554	4024	4462	4994	5009	6634
Average time for inner product in microseconds (10^{-6})										
Flat	9	23	38	80	×	×	×	×	×	×
Factored	8	8	11	11	14	16	14	25	65	55
Independent	14	21	38	42	97	158	230	396	1354	1419
Average time for backup computation in milliseconds (10^{-3})										
Flat	10705	41590	168074	672961	×	×	×	×	×	×
Factored	33	44	112	143	126	214	246	395	2836	1905
Independent	123	170	278	351	640	1822	2339	3494	44943	10915
Average belief state size (number of ADD vertexes)										
Factored	10	12	11	11	12	14	14	15	15	16
Independent	22	25	24	25	25	32	34	36	32	39
Average α-vector size (number of ADD vertexes)										
	389	493	909	1008	973	1759	1515	2181	5682	3187
Total time until convergence in seconds										
Flat	501	1862	35883	136544	×	×	×	×	×	×
Factored	1	1	5	17	17	26	33	590	2258	23751
Independent	14	17	113	224	214	243	370	1363	4774	35343
Final Average Discounted Reward (ADR) - computed over 200 trials										
	12.52	14.44	15.69	16.9	18.64	20.64	21.09	23.95	25.01	28.26

Table 4.6: Average execution time for basic operations on the rock sample problem with an 8×8 board and an increasing number of rocks.

n over m board containing k rocks. Each rock can contain some interesting substance or not. The locations of the rocks are known a priori but it is unknown whether they contain anything.

The agent has a long range sensor allowing it to check whether a rock is interesting from afar. Sensor accuracy reduces when the distance to the rock increases. The agent can move (deterministically) in each of the four directions, activate the long range sensor towards one of the rocks, or attempts to sample a rock.

Sampling an uninteresting rock, or sampling where there is no rock results in a cost of 10. Sampling an interesting rock results in a reward of 10. The execution terminates when there are no more interesting rocks.

The factored representation has state variables for X and Y coordinates and a state variable for each rock. There are 4 move actions, k sensor actions, and a single sample action. There are two observations – 'good' and 'bad'.

Network Administrator – In this domain a network administrator should maintain a network of n machines arranged in a ring topology (Guestrin et al., 2001). Each machine can be either up or down.

The administrator can ping a machine, checking whether the machine is up or down. The

Machines	6	7	8	9	10	11	12
$\|S\|$	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Average time for $g^{\alpha}_{a,o}$ computation in milliseconds (10^{-3})							
Flat	0.5	2	8	34	142	477	1827
Factored	1	1.9	5.1	14.9	36.7	110	227
Independent	5	1.9	5.1	14.9	36.7	110	227
Average time for τ computation in milliseconds (10^{-3})							
Flat	0.33	1.1	4.4	16.9	76.8	218	990
Factored	6.3	10.8	15.1	27.4	75.5	149	527
Independent	6.3	10.8	15.1	27.4	75.5	149	527
Average time for product in microseconds (10^{-6})							
Flat	34.1	36.1	45.6	66.2	93	172	373
Factored	76.1	131	226	305	481	647	820
Average time for backup computation in seconds							
Flat	0.11	0.68	10.8	81.4	452	1508	$\times$
Factored	0.8	3.8	15.4	52.8	89.1	362	599
Total time till convergence in seconds $\times 10^3$							
Flat	0.11	0.65	2.4	10.2	25.1	317	$\times$
Factored	0.22	1.06	3.5	11.3	24.7	110	480

Table 4.7: Average execution time for basic operations on the Network Administration problem with a ring configuration and different numbers of machines.

result has an accuracy of 0.95. A machine that is not working can be activated using the restart command. A machine may stop functioning with a probability of 0.05, but if one of its neighbors is already down, the probability increases to 0.33.

The administrator receives a reward of 1 for each working machine. There is a single special server machine, for which the administrator receives a reward of 2. A restart costs 2.5 and a ping costs 0.1.

The factored representation has n variables specifying the machine state. The agent can execute a ping or restart action for each machine ($2n$ actions) or a no-op action that does nothing. There are 2 observations — up and down.

Logistics – In the logistics domain the agent must distribute a set of n packages between m cities. Each package starts in a random city and has a predefined destination city. To move packages between cities the packages must be loaded onto a truck. The agent has k trucks it can use.

Each action can have stochastic effects — loading and unloading a package may fail with a probability of 0.1. Loading always fails if the specified truck and package are not at the same city. Driving the truck to a city may fail with a probability of 0.25, causing the truck to end up in some other city with uniform probability. Each action returns an observation of success or fail with 0.9 accuracy.

The agent can also ping a package or a truck to find its location. When pinging a truck the result is the truck location; when pinging a package the result is the truck it is on or the city it is at, if it is currently unloaded. Results have a 0.8 accuracy.

Driving the trucks between cities costs 1 and all other operations cost 0.1. When delivering a package to its final destination the agent receives a reward of 10.

For each truck there is a variable with m possible values (cities) and for each package there is a variable with $m+k$ possible values (cities and trucks). There are $n \times k$ load actions, loading a specific package to a specific truck, and n unload actions. There are $n + k$ ping actions.

Observations vary between actions, but there are at most $m + k$ observations.

Packages	2	3	4	5	6
$\|S\|$	2^{10}	2^{13}	2^{16}	2^{19}	2^{22}
Average time for τ computation in milliseconds (10^{-3})					
Flat	37	1539	36024	×	×
Factored	2.3	22.3	13.4	82.1	94.3
Average time for $g^{\alpha}_{a,o}$ computation in milliseconds (10^{-3})					
Flat	94	7393	537895	×	×
Factored	1.6	17.1	15.4	14.9	35
Inner product computation in microseconds (10^{-6})					
Flat	62.8	254.1	1188	×	×
Factored	95.5	855.2	487.9	3151	5428
Average time for backup computation in seconds					
Flat	2.8	80.3	5831	×	×
Factored	1.5	2.2	3.0	7.6	9.2
Total time till convergence in seconds					
Flat	203	15448	×	×	×
Factored	36	91	650	1559	5832

Table 4.8: Average execution time for basic operations on the Logistics domain with 4 cities, 2 trucks, and an increasing number of packages.

Results

We compare our ADD-based operations, using all the improvements explained above over all domains. In the tables below, Flat refers to representing belief states and α-vectors as mapping from states to values. In both belief states and α-vectors only non-zero entries are maintained. Factored denotes the ADD based representation.

	ADD size for Move actions	ADD size for Check actions	Average time for G computation
	7 Rocks - $\|S\| = 2^{13}$		
XX'	21790	107412	125
Mixed	51	1712	73
Relevant	18	494	4
	8 Rocks - $\|S\| = 2^{14}$		
XX'	44164	219506	330
Mixed	51	1994	145
Relevant	18	505	13
	9 Rocks - $\|S\| = 2^{15}$		
XX'	86536	446238	855
Mixed	54	2280	331
Relevant	18	525	44

Table 4.9: Influence of variable ordering and relevant variables over ADD size and computation time in the RockSample problem with an 8×8 board and increasing number of rocks. XX' —- first all pre-action variables and then all post-action variables. Mixed — X'_i specified immediately after X_i. Relevant — using only relevant variables in a mixed order.

Over all benchmarks the flat executions (which were the slowest) were stopped after a 48-hour timeout, resulting in an $\times$ symbol in the tables below. Over all benchmarks the flat representation and the ADD representation resulted in identical ADRs. The approximate belief state worked well for the Network Management and the RockSample domain, but failed on the Logistics domain.

Rocks	7	8	9
g-based	24,873	35,285	71,076
τ-based	216	290	408

Table 4.10: CPU time for a backup operation comparing g-based vs. τ-based backups over the RockSample 8×8 domain with relevant variables and increasing number of rocks.

The best results are presented in the Logistics domain as it displays the maximal action effect locality. The size of the relevant action diagrams is therefore compact and all point based operations are computed very rapidly.

Table 4.9 shows the influence of variable ordering and the elimination of irrelevant variables over ADD size and operation time for actions with many irrelevant variables (move actions in RockSample) and actions with a small number of irrelevant variables (check actions). Table 4.10 shows the advantage of τ-based backups compared to the standard g-based backup process used by point-based algorithms.

As we can see, in the RockSample and Logistics domains, ADD operations do not exhibit the exponential growth of operation time given the number of state variables. The inner product operation benefits the most from our improvements. As we can see, the belief states remain very structured and thus compact and, as a result, the size of the α-vectors does not have a considerable effect on the inner product. This is very important, as point-based algorithms execute a large number of inner products.

Our results clearly show the benefits of using ADDs for scaling up POMDP solvers on domains that present structure and effect locality. In practice, many real world domains possess these two properties. As such, we believe that ADD operations move POMDP research one step closer towards solving real world applications.

Symbolic Perseus

We attempted to execute Symbolic Perseus over our domains. Comparing our methods and Symbolic Perseus is difficult due to the various approximations it includes. In fact, Symbolic Perseus only managed to obtain competitive results over small sized problems (RockSample with a 4×4 board and the Network domain with up to 12 machines). For larger domains (RockSample 8×8 and logistics), Symbolic Perseus produced policies that resulted in a very low ADR.

To compare our approach to Symbolic Perseus we replaced FSVI with our own implementation of Perseus and executed it on the RockSample 4×4 instances on which Symbolic Perseus can attain comparatively good ADR levels. In these instances, we executed Symbolic Perseus until it produced its best policy and compared its best ADR against the average ADR of our ADD-based Perseus algorithm. The results are shown in Table 4.11. We can see that as the number of state variables increases, Symbolic Perseus's ADR decreases relative to our ADD-based Perseus. On larger domains, Symbolic Perseus was unable to obtain reasonable policies, whereas the ADD-based Perseus generated policies with substantially larger ADR. All results were obtained over the same machine, and both implementations use Java.

Rocks	2	3	4	5	6	7
SP maximal ADR	7.7 (4/5)	11.3 (4/5)	14.8 (2/5)	15.7 (1/5)	12.9 (2/5)	12.2 (2/5)
SP average time	143	678	820	1014	1667	556
ABP average best ADR	7.7	10.8	13.6	16.5	17.8	18.9
ABP average time	2	7	10	37	25	88

Table 4.11: Comparing Symbolic Perseus (SP) and our ADD-Based Perseus (ABP) implementation over the RockSample domain with a 4×4 board. In parentheses we show the portion of the cases where SP achieved the best result. We executed 5 runs with 1000 belief points and up to 30 iterations. Additional belief points or iterations did not increase the ADR.

Belief State Approximation

The reasons for Symbolic Perseus's failure on larger domains are unclear. We believed that our improved ADD-based techniques would be faster because of its faster implementation of various operations. However, these differences cannot account for such differences in the generated policies, so we were surprised to see that Symbolic Perseus was unable to generate reasonable policies. The reason for this must stem from additional approximations introduced by Symbolic Perseus. The most notable one is the belief state approximation method it employs. To better understand its influence on the computation we adapted this approximation in our code. Thus, we replaced the fully specified belief state ADDs in our implementation with a set of belief ADDs over the distinct state variable components. Each component is composed of the binary state variables that correspond to the same real, multi-value, state variable.

Table 4.6 shows the results of the belief state approximation as a product of independent components. The results we obtained are surprising. The good news is that belief-state approximation in our domains does not seem to adversely affect the quality of the policies obtained. The bad news is that belief-state approximations do not seem to provide any benefits in terms of space or time in our domains.

Belief state approximation is designed to limit the size of the belief state representation and to speed up the computation of belief related operations. However, over the RockSample domain, the approximation resulted in larger belief states and in slower belief update and inner product operations. Note that the algorithm is required to compute an ADD representing the entire belief state at each step. This takes time, and the resulting ADD is larger than the ADD we construct while utilizing variable relevance. Thus, it appears that our relevant variables improvement pays off here, and it may also be useful for many other applications, such as in the conformant planning community and in other applications where belief monitoring is used.

Given the fact that belief-state approximation does not decrease the ADR, we can only conclude that Symbolic Perseus fails to generate reasonable policies in larger domains mainly due to the limitations of the Perseus algorithm and the use of a bounded value function.

Chapter 5

Conclusions

This dissertation has two separate parts, focusing on two different aspects of acting under partial observability — learning a model of the environment and solving the resulting model. We present a number of important insights in both areas, as well as a number of algorithms that leverage these insights.

Research in the field of reinforcement learning under partial observability is traditionally split into two opposing approaches. The model-based approach learns a POMDP model of the environment and then solves it. The model-free approach learns only some internal state space and computes a solution (policy) directly, avoiding the learning of the environment dynamics. Research in model-free methods has always shown faster convergence to good policies, compared to model-based algorithms that were usually constructed using the slow Baum-Welch algorithm.

We show, however, that much of the success of model-free algorithms was due to the lack of observation noise. When observation noise is non existent or minor, computing an internal state space reduces the problem to the fully observable case. Therefore, model-free methods designed for MDPs can compute good policies. Unfortunately, as sensor noise increases, the performance of model-free methods reduces.

Nevertheless, we still wish to exploit the swiftness of model-free methods. We augment the agent with a sensor model that provides sensor accuracy information. We explain how, leveraging the sensor model, we can use any model-free method to construct a POMDP. This POMDP, though not perfect, still better captures the value of information and therefore provides superior policies. We continue to demonstrate how a well-known model-free method — McCallum's USM — can be transformed from a model-free technique into model-based, thus providing an online POMDP learning algorithm.

As such, this research brings closer together model-free and model-based techniques. It allows us to leverage both the rapid convergence of model-free methods and the enhanced expressiveness, and hence, superior policies, of model-based methods. It also demonstrates the need for testing new algorithms under noisy sensors, when evaluating their applicability.

Research in this field is still very far from applicability to real-world learning tasks. A step forward can be achieved by looking into the pruning of the internal state space to provide more compact POMDPs. It is also likely that a factored representation can be learned faster than the current flat model. We believe that further research into these techniques will allow us to move to larger, more interesting domains.

The complementary field of research, focusing on scaling up POMDP solvers to handle larger domains, has received much attention in the past decade. The new point-based approach has shown the ability to compute an approximate policy for larger domains than were ever handled before. In this area we also provide both an important observation for the creation of new point-based methods and stronger algorithms, able to handle larger domains.

We first show the effect of good order of value function updates (backups) over the convergence of the value function. Past research has focused on the selection of good belief points, but we show that a good sequence of backups can require an order of magnitude fewer operations. We show how well known point-based algorithms can benefit from our method of prioritizing backups, allowing them to solve problems much faster. We also present a new point-based algorithm that leverages the prioritization technique to solve problems faster, and using fewer backups than other point-based algorithms.

We observe, however, that point-based techniques that constantly maintain a large set of belief points have little hope of scaling up to larger domains. In such domains the storage requirement for remembering these belief points makes such algorithms impractical. We hence turn our attention to trial-based algorithms that only maintain a small number of belief points at each iteration. We present a new trial-based algorithm that has a remarkably rapid heuristic for finding good trajectories. Our algorithm is therefore able to handle all scalable benchmarks, as long as the system dynamics description fits into memory.

The traditional, flat description of the POMDP model can no longer support the scaling up of the solution algorithms. The rapid algorithms we suggest can solve models that can no longer fit into memory. A factored representation of the POMDP can describe much larger models efficiently. Such a representation usually uses Dynamic Bayesian Networks and Algebraic Decision Diagrams to describe the environment dynamics. Our implementation of all point-based operations using ADDs scales up to larger domains than was considered possible.

The combined effect of the solving techniques is now able to compute approximate policies for domains that are highly stochastic, with considerable sensor noise, and with millions of states, as our experiments with the Logistics domains demonstrate. Considering that only a decade ago POMDP solvers could handle only problems with less than 10 states, this advancement is indeed remarkable.

Factored representations also provide a rich ground for additional improvements. We currently restrict ourselves to factored states, yet it is likely that a factored representation of actions and observations will scale up better. New algorithms need to be developed for combining the factored actions into a complete action optimally or approximately. Also, efficient operations that consider the possible combinations of the factored observations are needed. We believe that continuing work in this area will bring us a few steps closer to our true goal — the ability to handle real-world tasks under partial observability.

Bibliography

Aberdeen, D. (2003). A (revised) survey of approximate methods for solving partially observable markov decision processes. Tech. rep., National ICT Australia, Canberra, Austalia.

Alexandrov, S. (2003). Ratbert: Nearest sequence memory based prediction model applied to robot navigation. In *Proceedings of the 20th International Conference on Machine Learning*.

Baird, L., & Moore, A. (1998). Gradient descent for general reinforcement learning. In *Proceedings of the 11th International Conference on Neural Information Processing Systems*, pp. 968–974. MIT Press.

Baird, L. C. (1999). *Reinforcement Learning Through Gradient Descent*. Ph.D. thesis, Carnegie Mellon University.

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Inteligence*, *72(1)*, 81–138.

Bellman, R. E. (1962). *Dynamic Programming*. Princeton University Press.

Bilmes, J. (1997). A gentle tutorial on the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models. Tech. rep. ICSI-TR-97-021, Barkley, CA.

Binder, J., Koller, D., Russell, S. J., & Kanazawa, K. (1997). Adaptive probabilistic networks with hidden variables. *Machine Learning*, *29*(2-3), 213–244.

Bonet, B., & Gefner, H. (1998). Solving large POMDPs using real time dynamic programming. In *AAAI Fall Symposium on POMDPs*.

Boutilier, C., & Poole, D. (1996). Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pp. 1168–1175 Portland, Oregon, USA. AAAI Press / The MIT Press.

Boyen, X., & Koller, D. (1998). Tractable inference for complex stochastic processes. In *UAI '98*, pp. 33–42.

Brafman, R. I. (1997). A heuristic variable grid solution method for POMDPs. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pp. 76–81 Providence, Rhode Island. AAAI Press / MIT Press.

Braziunas, D., & Boutilier, C. (2004). Stochastic local search for pomdp controllers. In *Proceedings of The Nineteenth National Conference on Artificial Intelligence (AAAI-04)*. Morgan Kaufman.

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, *35*(8), 677–691.

Cassandra, A., Littman, M. L., & Zhang, N. L. (1997). Incremental pruning: A simple, fast, exact algorithm for partially observable markov decision processes. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence*, pp. 54–61. Morgan Kaufmann.

Cassandra, A. R., Kaelbling, L. P., & Littman, M. L. (1994). Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Vol. 2, pp. 1023–1028. AAAI Press/MIT Press.

Cheng, H. T. (1988). *Algorithms for partially observable Markov decision processes*. Ph.D. thesis, University of British Columbia.

Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *National Conference on Artificial Intelligence AAAI*, pp. 183–188.

Estelle, J. (2003). Reinforcment learning in pomdps: Instance-based state identification vs. fixed memory representations. Tech. rep., Cornell University.

Guestrin, C., Koller, D., & Parr, R. (2001). Solving factored pomdps with linear value functions. In *IJCAI workshop on Planning under Uncertainty and Incomplete Information*.

Hansen, E. A. (1998). Solving pomdps by searching in policy space. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pp. 211–219. Morgan Kaufmann.

Hansen, E. A., & Feng, Z. (2000). Dynamic programming for POMDPs using a factored state representation. In *Artificial Intelligence Planning Systems (AIPS)*, pp. 130–139.

Hauskrecht, M. (2000). Value-function approximations for partially observable markov decision processes. *Journal of Artificial Intelligence Research (JAIR)*, *13*, 33–94.

Hayashi, A., & Suematsu, N. (1999). Viewing Classifier Systems as Model Free Learning in POMDPs. In *Advances in Neural Information Processing Systems*, pp. 989–995.

Hernandez, N., & Mahadevan, S. (2000). Hierarchical memory-based reinforcement learning. In *Proceedings of the Fifteenth International Conference on Neural Information Processing Systems*, pp. 1047–1053. Morgan Kaufmann.

Hoey, J., von Bertoldi, A., Poupart, P., & Mihailidis, A. (2007). Assisting persons with dementia during handwashing using a partially observable markov decision process. In *International Conference on Vision Systems (ICVS)*.

Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. MIT Press.

Izadi, M. T., Precup, D., & Azar, D. (2006). Belief selection in point-based planning algorithms for pomdps.. In *Canadian Conference on AI*, pp. 383–394.

Jaakkola, T., Singh, S. P., & Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable Markov decision problems. In Tesauro, G., Touretzky, D., & Leen, T. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 7, pp. 345–352. MIT Press.

James, M., & Singh, S. (2004). Learning and discovery of predictive state representations in dynamical systems with reset. In *Proceedings of the 21st International Conference on Machine Learning*, pp. 53–61. ACM Press.

James, M., Singh, S., & Littman, M. (2004). Planning with predictive state representations. In *Proceedings of the 21st International Conference on Machine Learning and Applications*, pp. 304–311.

Kaelbling, L. P., Littman, M., & Moore, A. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

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

Lanzi, P. L. (2000). Adaptive Agents with Reinforcement Learning and Internal Memory. In *Proceedings of the Sixth International Conference on the Simulation of Adaptive Behavior*, pp. 312–322. MIT Press.

Lin, L.-J., & Mitchell, T. M. (1992a). Memory approaches to reinforcement learning in non-markovian domains. Tech. rep. CMU-CS-92-138, Carnegie Mellon University, Pittsburgh, PA 15213.

Lin, L. J., & Mitchell, T. M. (1992b). Reinforcement learning with hidden states. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pp. 271–280. MIT Press.

Littman, M. L. (1994). Memoryless policies: Theoretical limitations and practical results. In Cliff, D., Husbands, P., Meyer, J.-A., & Wilson, S. W. (Eds.), *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pp. 238–245 Cambridge, MA. MIT Press.

Littman, M. L., Cassandra, A. R., & Kaelbling, L. P. (1995a). Efficient dynamic-programming updates in partially observable markov decision processes. Tech. rep. CS-95-19, Brown University.

Littman, M. L., Cassandra, A. R., & Kaelbling, L. P. (1995b). Learning policies for partially observable environments: Scaling up. In Prieditis, A., & Russell, S. (Eds.), *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 362–370 San Francisco, CA, USA. Morgan Kaufmann publishers Inc.

Loch, J., & Singh, S. (1998). Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes. In *Proceedings 15th International Conference on Machine Learning*, pp. 323–331. Morgan Kaufmann, San Francisco, CA.

Lovejoy, W. S. (1991). Computationally feasible bounds for partially observable markov decison processes. *Operations Research*, *39*, 175–192.

McCallum, A. K. (1993). Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 190–196.

McCallum, A. K. (1995). Instance-based state identification for reinforcement learning. *Advances in Neural Information Processing Systems*, *7*, 377–384.

McCallum, A. K. (1996). *Reinforcement Learning with Selective Perception and Hidden State*. Ph.D. thesis, Department of Computer Science, University of Rochester.

Meuleau, N., Kim, K. E., Kaelbling, L. P., & Cassandra, A. R. (1999a). Solving pomdps by searching the space of finite policies. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pp. 417–426. Morgan Kaufmann.

Meuleau, N., Peshkin, L., Kim, K., & Kaelbling, L. P. (1999b). Learning finite-state controllers for partially observable environments. In *Proceedings of the Fifteenth Conference on Uncertainity in Artificial Intelligence*, pp. 427–436. Morgan Kaufmann.

Moore, A. W., & Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, *13*, 103–130.

Nikovski, D. (2002). *State-Aggregation Algorithms for Learning Probabilistic Models for Robot Control*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

Nikovski, D., & Nourbakhsh, I. (2000). Learning probabilistic models for decision-theoretic navigation of mobile robots. In *Proceedings of the 17th International Conference on Machine Learning*, pp. 671–678. Morgan Kaufmann, San Francisco, CA.

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

Paquet, S., Tobin, L., & Chaib-draa, B. (2005). An online pomdp algorithm for complex multiagent environments. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pp. 970–977. ACM Press.

Parr, R., & Russell, S. (1995). Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1088–1094.

Parr, R., & Russell, S. (1997). Reinforcement learning with hierarchies of machines. In *In Proceedings of Advances in Neural Information Processing Systems*, Vol. 10, pp. 1043–1049. MIT Press.

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

Pineau, J., Gordon, G., & Thrun, S. (2003). Point-based value iteration: An anytime algorithm for POMDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1025–1032.

Pineau, J., Gordon, G., & Thrun, S. (2006). Anytime point-based approximations for large pomdps. *Journal of Artificial Intelligence Research (JAIR)*, *27*, 335–380.

Poupart, P. (2002). *Partially Observable Markov Decision Processes*. Ph.D. thesis, University of Toronto.

Poupart, P., & Boutilier, C. (2002). Value directed compression of pomdps. In *Proceedings of the 15th International Conference on Neural Information Processing Systems*, pp. 1547–1554.

Poupart, P., & Boutilier, C. (2004). Bounded finite state controllers. In *Proceedings of the 16th International Conference on Neural Information Processing Systems*. MIT Press.

Poupart, P., Vlassis, N., Hoey, J., & Regan, K. (2006). An analytic solution to discrete bayesian reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, pp. 697–704. ACM Press.

Puterman, M. (1994). *Markov Decision Processes.* Wiley, New York.

R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, & F. Somenzi (1993). Algebraic Decision Diagrams and Their Applications. In *IEEE /ACM International Conference on CAD*, pp. 188–191 Santa Clara, California. IEEE Computer Society Press.

Rosencrantz, M., Gordon, G., & Thrun, S. (2004). Learning low dimensional predictive representations. In *Proceedings of the Twenty-First International Conference on Machine Learning*, pp. 88–96 Banff, Alberta, Canada.

Ross, S., & Chaib-draa, B. (2007). Aems: An anytime online search algorithm for approximate policy refinement in large pomdps. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 2592–2598.

Shani, G., & Brafman, R. I. (2004). Resolving perceptual aliasing in the presence of noisy sensors. In *Proceedings of the Eighteenth Annual Conference on Neural Information Processing Systems*. MIT Press.

Shani, G., Brafman, R. I., & Shimony, S. E. (2005a). Model-based online learning of pomdps. In *Proceedings of the European Conference on Machine Learning (ECML)*, Vol. 3720 of *Lecture Notes in Computer Science*, pp. 353–364. Springer.

Shani, G., Brafman, R. I., & Shimony, S. E. (2005b). Partial observability under noisy sensors - from model-free to model-based. In *RRfRL Workshop, International Conference on Machine Learning (ICML)*.

Shani, G., Brafman, R. I., & Shimony, S. E. (2006). Prioritizing point-based pomdp solvers. In *Proceedings of the European Conference on Machine Learning (ECML)*, Vol. 4212 of *Lecture Notes in Computer Science*, pp. 389–400. Springer.

Shatkay, H. (1999). Learning hidden markov models with geometrical constraints. In *In Proceedings of the International Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 602–611.

Singh, S., James, M. R., & Rudary, M. R. (2004). Predictive state representations: A new theory for modeling dynamical systems. In *Proceedings of the Twentieth International Conference on Uncertainty in Artificial Intelligence*, pp. 512–519.

Singh, S., Littman, M. L., Jong, N. K., Pardoe, D., & Stone, P. (2003). Learning predictive state representations. In *Proceedings of the Twentieth International Conference on Machine Learning*, pp. 712–719.

Singh, S., Littman, M. L., & Sutton, R. S. (2001). Predictive representations of state. In *Proceedings of the 15th International Conference on Neural Information Processing Systems*, pp. 1555–1561. MIT Press.

Singh, S. P., & Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, *22*(1–3), 123–158.

Smith, T., & Simmons, R. (2004). Heuristic search value iteration for pomdps. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence (UAI)*, pp. 520–527. AUAI Press.

Smith, T., & Simmons, R. (2005). Point-based pomdp algorithms: Improved analysis and implementation. In *Proceedings of the 21st conference on Uncertainty in artificial intelligence (UAI)*, pp. 542–549. AUAI Press.

Sondik, E. J. (1971). *The Optimal Control of Partially Observable Markov Processes*. Ph.D. thesis, Stanford University.

Spaan, M. T. J., & Vlassis, N. (2005). Perseus: Randomized point-based value iteration for POMDPs. *Journal of Artificial Intelligence Research (JAIR)*, *24*, 195–220.

St-Aubin, R., Hoey, J., & Boutilier, C. (2000). APRICODD: Approximate policy construction using decision diagrams. In *Advances in Neural Information Processing Systems(NIPS)*, pp. 1089–1095. MIT Press.

Suematsu, N., & Hayashi, A. (1999). A reinforcement learning algorithm in partially observable environments using short-term memory. *Advances in Neural Information Processing Systems*, *11*, 1059–1065.

Suematsu, N., Hayashi, A., & Li, S. (1997). A Bayesian approach to model learning in non-Markovian environments. In *Proceedings of the 14th International Conference on Machine Learning*, pp. 349–357. Morgan Kaufmann.

Sutton, R. S. (1991). Planning by incremental dynamic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 353–357. Morgan Kaufmann.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

Washington, R. (1997). BI-POMDP: Bounded, incremental, partially-observable markov-model planning. In *ECP '97: Proceedings of the 4th European Conference on Planning*, pp. 440–451.

Watkins, C., & Dayan, P. (1992). Q-learning. *Journal of Machine Learning*, *8*, 279–292.

Whitehead, S. D., & Ballard, D. H. (1991). Learning to perceive and act by trial and error. *Journal of Machine Learning*, *7*, 45–83.

Wiering, M., & Schmidhuber, J. (1997). Hq-learning. *Adaptive Behavior*, *6*(2), 219–246.

Wierstra, D., & Wiering, M. (2004). Utile distinction hidden markov models. In *Proceedings of the Twenty-First International Conference on Machine Learning*, pp. 108–116.

Williams, J. K., & Singh, S. (1998). Experimental results on learning stochastic memoryless policies for partially observable markov decision processes. In *Proceedings 11th Conference on Neural Information Processing Systems*, pp. 1073–1079. The MIT Press.

Wingate, D., & Seppi, K. D. (2005). Prioritization methods for accelerating mdp solvers. *Journal of Machine Learning Research (JMLR)*, *6*, 851–881.

Yeh, A. (2000). More accurate tests for the statistical significance of result differences. In *Proceedings of the Eighteenth International Conference on Computational Linguistics*, pp. 947–953.

Zhou, R., & Hansen, E. A. (2001). An improved grid-based approximation algorithm for POMDPs. In *Proceedings of the International Conference on Artificial Intelligence (IJCAI)*, pp. 707–716.