Point-Based POMDP Solvers: Survey and Comparative Analysis

Robert Kaplow

Master of Science

School of Computer Science

McGill University

Montreal, Quebec

2010-04-17

A thesis submitted to McGill University
in partial fulfillment of the requirements
of the degree of Master of Science

DEDICATION

To my family and friends.

ACKNOWLEDGEMENTS

There are many people I need to thank for all the support and guidance I have been given for this thesis, and throughout my time at McGill, both undergraduate and postgraduate. First and foremost, I need to thank my advisor Joelle Pineau, who I've had the pleasure of working with for several years now. Whether providing me with new research ideas, helping me analyze new results, and being able to make the most complicated problems simple, her contribution was invaluable, and without which none of this would have been possible. I need to thank Guy Shani for providing me with a wonderful internship opportunity and providing much of the research basis for this thesis, as well as being always there for questions. Many thanks to Amin Atrash for acting as my mentor throughout these past few years, and thanks to Cosmin Paduraru for advice on my thesis as well as fun research diversions in Reinforcement Learning. I need to thank Doina Precup for her advice on earlier research projects. Thanks to Julien Villemure for his help with my thesis and many good times. Many thanks to Peter Sloan for his edits and comments, as well as for understanding more about this work than he should with no background. And thanks to my sister Lauren Kaplow for keeping my thesis legible, and enjoying stochasticity.

I have had a wonderful time working in the Reasoning and Learning Lab these past years, and I want to thank the people that made the lab what it was, Robert West, Arthur Guez, Marc Bellemare, Stephane Ross, Caitlin Phillips, Robert Vincent, Jordan Frank, Keith Bush, Hiba Yamani, Pablo Castro, Gheorge Comanici, Monica Dinculescu and many others. I want to thank all my friends from CTF for a wonderful undergraduate experience, and especially David Meunier for working with me so much during that time. Thanks so much to some of my oldest and dearest friends, without which these past few years would have been all the poorer: Zachary Lisak, Joshua Berger, Pamela Bernstein, Hayley Vininsky, Amanda Saxe, David Brenner, Jon Rubin, David Gomel, Richard Roskies and many others. Thank you.

Finally, thanks so much to my girlfriend Stephanie and my family: Howard, Marilyn and Lauren. I couldn't have done this without you.

ABSTRACT

Planning under uncertainty is an increasingly important research field, and it is clear that the design of robust and scalable algorithms which consider uncertainty is key to the development of effective autonomous and semi-autonomous systems. Partially Observable Markov Decision Processes (POMDPs) offer a powerful mathematical framework for making optimal action choices in noisy and/or uncertain environments. However, integration of the POMDP model with real world applications has been slow due to the high computation cost of exact approaches to POMDP planning.

In recent years, point-based POMDP solvers have emerged as efficient methods for providing approximate solutions by planning over a small subset of the belief space. This thesis first provides a survey on many of the proposed point-based POMDP solvers. We then conduct an empirical analysis on the key components of point-based methods, the belief collection and belief updating processes. This is an important contribution, as previous publications on point-based methods have only compared full algorithms, without comparing the underlying processes. As well, we verify the effect of a variety of parameters and optimizations that could be used within a point-based solver. Experiments are conducted on a variety of POMDP environments.

ABRÉGÉ

L'importance grandissante de la recherche dans le domaine de la planification sous incertitude est signe que l'élaboration d'algorithmes robustes et extensibles qui gèrent l'incertitude est un élément clé dans le développement de systèmes autonomes et semi-autonomes efficaces. Les processus de décision markoviens partiellement observables (POMDP) constituent une puissante fondation mathématique pour le choix d'actions optimales dans un environnement incertain. Il a cependant été difficile d'incorporer les POMDPs à des applications réelles, à cause de leur coût de calcul élevé lorsqu'une solution exacte est requise.

Récemment, les approches de résolution de POMDPs dites par points, qui planifient sur un petit sous-ensemble de l'état des croyance, se sont révélées être efficaces pour obtenir des solutions approximatives. Le présent mémoire propose tout d'abord une revue de plusieurs approches par points. Par la suite, une analyse empirique des composantes primordiales des approches par points, de la collecte d'observations, ainsi que du processus de mise à jour de l'état des croyance, est proposée. De plus, les effets de différents paramètres et optimisations liés aux approches par points sont vérifiés. Des expériences sont conduites avec une variété d'environnements de type POMDP.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Artificial Intelligence (AI) has been defined as "the study and design of rational agents" [40]. AI research has begun to permeate modern society, with sophisticated AI techniques commonly in use in health care, finance, internet search and robotics. As the use of AI technologies grows, it becomes increasingly important for these techniques to be robust and scalable.

For decades, planning has been a key component in AI research. We can define planning as "devising a plan of action to achieve one's goals" [45]. We will use the generic term *agent* to denote what or who carries out the plan of action. In practice, this could be a software system, a robot, or even instructions to a human operator. It is clear that planning is required for any complex autonomous system. Even partially-autonomous or expert systems still require robust planning.

In this work, we focus on the subdomain of planning known as sequential decision making. In this task, the agent must perform a sequence of decisions while trying to accomplish some planning goal, or reach some measure of performance. In a general sequential decision making task, the agent must learn to balance between greedily accomplishing short term goals, with planning for longer term goals. This mindset separates the planning field from problem solving where only a single decision must be made, such as in utility theory, decision networks or simple expert systems.

AI techniques are increasingly used in real world conditions, especially in robotics and medical domains. These types of environments are much more difficult to plan for, since the real world is inherently noisy. There are many reasons why: robotic sensors such as laser rangefinders can give poor estimates due to a dynamic environment or inherent hardware limitations, a speech recognizer can give an incorrect text output due to slurred speech or substantial background noise, or a medical measurement can give confusing readings due to normal human variance in the measurement. Uncertainty can manifest itself in a variety of ways: in the state of the environment (for example, in a medical domain, the problem with the patient may be unclear), in what a specific action may do (for example, a robot attempting to turn in a tight arc may slip), or in the information the agent receives (the speech recognizer is not sure what word the user has uttered).

We now describe the different types of uncertainty. An environment is *probabilistic* or *stochastic* if there is uncertainty in what a specific decision will do to the environment (as in the example above of a robot slipping when it turns). A stochastic environment is generally more difficult to plan in than a deterministic environment, since your decisions must be taken knowing that their results are not predictable.

We refer to an environment as *partially observable* if the current state of the environment (including the state of the agent) is unknown in some way. This is the case when the information from our sensors does not fully describe the situation, and the agent cannot therefore pinpoint its abstracted spacial representation in the environment. This is, unfortunately, all too common in real world domains, such as in robotics navigation and dialogue management.

Partial observability is generally a more challenging form of uncertainty than the uncertainty provided by a stochastic domain. A major challenge posed by partial observability is that the uncertainty builds throughout the planning. If the algorithm ignores the fact that the environment is highly uncertain, the agent can quickly lose track of the situation and make poor decisions. This means that an agent acting in a partially observable environment must be able to incorporate information from the history of the task, which can include previous sensor values and its past decisions.

All of these possible uncertainties seem to pose a daunting challenge to the algorithm designer, and it may seem impossible to design a robust solution when there are a multitude of uncertainties in the task environment. The key to designing a solution to this problem is for the agent to not only be aware of its own uncertainty, but to *use the knowledge of its own uncertainty as part of the solution*. This is a critical step for planning in partially observable environments. An agent which makes decisions knowing that it has a poor knowledge of the current situation, and acting accordingly, will do much better than an agent which simply tries to make the best decision while ignoring its own uncertainty. By this, we mean an agent should be able to make decisions which obtain information about itself and the environment. An agent should choose to gather information about the environment if doing so will lead a better long term goal. However, there is often a cost to gathering information, so the agent must balance gathering information and attempting to solve the problem. In this work, we focus on examining a variety of techniques to intelligently solve partially observable problems, which provide the agent with information on how to act optimally or near-optimally under uncertainty.

1.1 Planning

AI planning originated from control theory and more classic search methods as well as a more formal basis in theorem proving. The field was created as a response to the increasing need for intelligent techniques for scheduling tasks, robotics, and other domains [45]. Early classical planning focused on extremely high level abstractions of real environments. The speed of the computers at the time forced a simpler representation. The domains tended to be small and finite, as well as being deterministic and fully observable. The STRIPS [17] planner was an early example of these kinds of solvers, used for robot control. This type of planning encoded the task in terms of logical statements such a pre- and post- conditions, and formulated the plan via a state-space search system. The next phase in planning research focused on partial-order planning, which specifies a plan based on which actions must go before which other actions (a partial ordering over the possible actions of the agent). These can be created through a search though plan-space. Examples of these planners include SNLP [50] and UCPOP [37]. Other methods such as planning graphs [10] and decision diagrams [15] have been used for planning (see [2, 56] for surveys on classic planning). While these planning methods have become increasingly sophisticated, they can encounter difficulties when applied to real-world planning domains, since they do not model the stochastic and uncertain nature of realistic environments.

Another major influence in the development of modern decision making is the field of dynamic programming. Dynamic programming encompasses many techniques to solve problems by breaking them down into subproblems, and exploiting overlapping structures of these subproblems. Richard Bellman's book "Dynamic

Programming" [9] described many of the original algorithms, and provided a base for the development of the field. Howard expanded on many of these ideas [22]. The Markov Decision Process (MDP) framework, which was used in the original dynamic programming work has become a very popular model for probabilistic decision making, due to its generality and ability to describe stochastic domains. The expressible stochasticity of the MDP framework allows it to capture the uncertainty in future actions. The related field of reinforcement learning [53] focuses on developing methods to learn unknown, possibly stochastic, environments via exploration.

The MDP framework is able to model stochastic environments, but it assumes that the agent is totally aware of the status of the environment itself (i.e. that the environment is fully observable). For example, it could not model a situation with hidden information, such as an opponent's cards in poker or the current severity of a disease with a patient. The Partially Observable Markov Decision Process (POMDP) [3], the natural partially observable extension to MDPs, has emerged as a flexible model for decision making under uncertainty. Sondik [51] first proposed an exact method for solving a POMDP. More efficient exact solving methods have been proposed [14, 16, 27, 58, 59]. In recent years, there has been much work in developing approximate solutions. In this thesis, we will focus on a class of approximate POMDP solvers known as point-based methods.

Point-based POMDP planning algorithms create approximate POMDP solutions by only considering a finite subset of points in belief space. There have been many [26, 38, 39, 46, 48, 52] such algorithms proposed in the past few years. While each algorithm has been compared with predecessors, there are a few problems with these comparisons. A primary issue is that these algorithms tend to introduce several new ideas at once. While effective at pushing the state of the art in POMDP solving, this makes it more difficult to compare the approaches. This approach to algorithm design makes it especially difficult to isolate the element that produced the performance increase. To date, there has not been a comprehensive study examining the methods introduced in these algorithms.

1.2 Contribution

The primary contribution of this thesis is to offer a comprehensive empirical analysis of point-based POMDP solvers. While previous comparisons have focused on contrasting the speed of convergence of different algorithms, in this work we focus on comparing the specific mechanics used in these algorithms. The advantage of framing the examination in terms of the methods used within the algorithms is that we can keep all algorithm parameters fixed, and gain a deeper understanding about which parts of the algorithms are useful, and in which context.

Point-based POMDP solvers also have several important parameters that need to be set, such as the size of the belief space and the number of updates per iteration. To our knowledge, there has not been a formal study showing the effect of these parameters on the quality of the POMDP solution. A primary contribution of this thesis is to examine the importance of these algorithm components. We also examine how other mechanics, such as better initial bounds, affect the quality of the solution.

The empirical performance of a given approximate POMDP solver depends highly on the domain, and can vary depending on the methods and the parameters of the algorithm. To this end, we applied our empirical examination on a wide set of target domains.

It is often the case that a careful study of current methods can give rise to new ideas. In the course of our study, we developed two modifications to a classic point-based technique. We combine these ideas to create the algorithm PBVI Leaf Biased Collection. In our investigations, we test the quality of this new approach.

1.3 Outline

We now present the overall organization structure of the thesis. In Chapter 2, we provide formal descriptions of the MDP and POMDP decision making frameworks, as well as introduce key concepts such as value functions and belief states. We then provide an overview of exact and approximate methods for solving POMDPs. We present a survey on the current point-based POMDP planning algorithms in Chapter 3, with a strong focus on what differs in these methods. In Chapter 4, we describe a variety of methods used by the point-based algorithms. We do an empirical evaluation of these methods and present our results and observations in Chapter 5. Finally, we conclude in Chapter 6.

CHAPTER 2

Probabilistic Decision Making in Markovian Environments

The concept of planning and decision making is central to many problems in AI. In a decision making process, an agent makes a set of choices by selecting from some set of possible actions, to achieve goals specified by the problem domain. In this work, we only consider discrete time control processes, i.e. an action must be selected at fixed time steps. The agent exists in a *state* of the environment, which is some encoding of the current properties of the environment. When formulating a problem domain, the problem designer should try to keep the state as simple as possible, as a very complicated state description would make solving the decision making process much more difficult. To achieve this, a common assumption to make is the Markov property. A process has the Markov property when the distribution over the future states of the process depends only on the current state and not past states. Finally, we consider stochastic decision making processes, where the next state is determined by some probability distribution rather than the special case of a deterministic process (with a fixed next state).

In this chapter, we describe a classic model for probabilistic decision making, the Markov Decision Process (MDP), as well as detail some methods for planning in an MDP. We then detail the partially observable extension to the MDP, the Partially

Observable Markov Decision Process (POMDP). Subsequently we describe existing POMDP planning methods, both exact and approximate.

2.1 Markov Decision Processes

2.1.1 Description

The Markov Decision Process (MDP) is a powerful model for solving stochastic decision making problems [9]. We can define a MDP as a tuple (S, A, T, R, γ), where we define:

States $S = \{s_0, s_1, \ldots, s_N\}$ is a finite set of all possible states of the agent/environment. We denote s_t as the state at timestep t. As we have discussed, the state is a sufficient statistic to fully describe everything the agent might need to form its decision. MDPs can also be defined with a continuous state space, but in this thesis we only consider the finite case. An example of a state is the location of the robot in a navigation task.

Actions $A = \{a_0, a_1, \ldots a_M\}$ is the finite set of all possible actions the agent can take (one at each time step). A continuous set of actions is also possible, but in this work we consider only a finite action space. In general, it is possible for the set of actions to depend on the state, but in this work we keep a fixed set of actions available in all states. An example of an action could be a move command in a navigation task.

Transition Function We define the transition function T as

$$T(s, a, s') = P(s_{t+1} = s' | s_t = s, a_t = a) \forall t, \tag{2.1}$$

where s is the state at time t, a is the action taken at time t, and s' is the state at time $t+1$. The transition function defines the next state probability distribution under a specific state-action pair. This property of the MDP models the stochastic nature of the decision making. The transition function satisfies the Markov property, i.e. $P(s_{t+1} = s'|s_t, a_t, s_{t-1}, a_{t-1}, ..., s_0, a_0) = P(s_{t+1} = s'|s_t, a_t)$. Since it is a full conditional probability distribution, we also have the property that $\sum_{s' \in S} T(s, a, s') = 1$.

Reward The function $R(s, a) \in \mathbb{R}$ is the numerical reward for taking action a in state s. The model includes both positive rewards (payoffs) and negative rewards (costs). The goal of any agent is to maximize the total return from the process, which we define as

$$E\left[\sum_{t=t_0}^{T} \gamma^{t-t_0} r_t\right], \tag{2.2}$$

where E is expectation, r_t is the reward earned at timestep t, γ is defined subsequently and t_0 and T are the start and end times of the process, respectively. The expectation E is simply the average return over an infinite number of trials.

Discount We define $\gamma \in [0, 1)$ as the discount factor. As we saw in Equation 2.2, the reward at timestep t is discounted by the factor $\gamma^{(t-t_0)}$. The intuition behind the discount factor is to value short term gains more strongly than longer term gains of equal size. If γ is close to 1, then the agent will value future rewards strongly. As γ approaches 0, the agent becomes much more myopic, maximizing more immediate rewards. The discount factor can be seen as an analogue to the interest rate in finance. As well, γ ensures that the total return is a finite sum.

For a specific task in a MDP, we also require a start state s_0, or some initial start state distribution $\mu_0(s) = P(s_0 = s)$, where $\sum_{s \in S} \mu_0(s) = 1$.

We define a *policy* $\pi \in \Pi$ as

$$\pi(s,a) = P(a_t = a | s_t = s). \tag{2.3}$$

The policy defines a probability distribution over which action to take based on the current state, i.e. $\sum_{a \in A} \pi(s,a) = 1 \; \forall s$. The policy is used by the agent to carry out the decision making task.

Bellman [9] showed that for a given MDP, there always exists a deterministic *optimal policy* π^*, which gives a long term return at least as good as any other policy $\pi \in \Pi$ starting in any state s. A deterministic policy is one where $\pi(s, a_i) = 1$ for a given action a_i, and $\pi(s,a) = 0 \quad \forall a \neq a_i$. Since there is no advantage in using stochastic policies, we will refer only to deterministic policies for simplicity. Therefore, we will only use policies of the form $\pi(s) \rightarrow a$, which will always pick action a in state s.

With the MDP and policy defined, we can trace the execution of an agent within an environment. The agent starts in the start state s_0. At each time step t, the agent selects an action $a_t = \pi(s_t)$. The agent executes a_t, moving into state s_{t+1} and receiving reward r_t. This process can either proceed indefinitely (continuous task), or finish under some termination condition (episodic task).

2.1.2 Planning

Value Functions and Bellman Equations

An important tool for finding a policy for an MDP is the notion of a *value function* $V^\pi(s)$. Formally, we define a value function V^π as

$$V^\pi(s) = E_\pi[R_t|s_t = s] = E_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k}|s_t = s\right]. \tag{2.4}$$

The intuition behind the value function of a state s is that it represents the total amount of reward the agent will receive if it started in state s and followed the associated policy π afterwards.

Similarly, we can define a state-action value function $Q^\pi(s, a)$ as

$$Q^\pi(s,a) = E_\pi[R_t|s_t = s, a_t = a] = E_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k}|s_t = s, a_t = a\right]. \tag{2.5}$$

$Q^\pi(s, a)$ is the expected return from starting in state s, taking action a and afterwards following the policy π.

An important property of the state value function V^π is that it satisfies the following recursive relationship:

$$V^\pi(s) = R(s,a) + \gamma\sum_{s'\in S} T(s,a,s')V^\pi(s'), \tag{2.6}$$

where $a = \pi(s)$. This follows the intuition that the value for a state s is the immediate reward added to the (weighted) discounted value of the next state.

We can now more formally define optimality for policies and value functions. We define an optimal policy π^*,

$$\pi^* = \operatorname*{argmax}_{\pi \in \Pi} V^\pi(s_0), \tag{2.7}$$

and the optimal value function,

$$V^*(s) = \max_{\pi \in \Pi} V^\pi(s). \tag{2.8}$$

Bellman showed [9] that the optimal value function satisfies the following recursive equations:

$$V^*(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right], \tag{2.9}$$

and similarly for $Q^*(s,a)$,

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') \max_{a'} Q^*(s',a'). \tag{2.10}$$

These are known as the Bellman equations [9] for the value function. The intuition is that the optimal value for a state s is the value received by taking the action which leads to a maximum combination of immediate reward and total future discounted return. It is clear we can extract an optimal policy π^* from the optimal value function V^*:

$$\pi^*(s) = \operatorname*{argmax}_{a} \left[R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right]. \tag{2.11}$$

Value Iteration

There are several classic dynamic programming methods that have been developed for solving for the optimal value function in MDP problem, including policy evaluation, policy iteration and value iteration [9, 22]. Value iteration [9] can be written as a simple backup operation:

$$V_{k+1}(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V_k(s') \right]. \tag{2.12}$$

The full algorithm initializes the value function, then for each state, updates the value based on the backup equation above. At each step, the value function gives a closer approximation to the optimal value function. This is repeated until some stopping condition, such as:

$$\max_{s \in S} |V_k(s) - V_{k-1}(s)| \leq \delta, \tag{2.13}$$

for some small $\delta \geq 0$.

At any step, a policy π_{k+1} can be computed directly from the value function V_k:

$$\pi_{k+1}(s) = \operatorname*{argmax}_{a} \left[R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V_k(s') \right]. \tag{2.14}$$

If value iteration is run until the above stopping condition is achieved, then $\max_{s \in S} |V^*(s) - V^{\pi_{k+1}}(s)| < \frac{2\gamma\delta}{1-\gamma}$. A policy π is defined to be ϵ-optimal if $\max_{s \in S} |V^*(s) - V^{\pi}(s)| < \epsilon$. Therefore, a policy resulting from value iteration is guaranteed to be ϵ-optimal if we execute value iteration until

$$\max_{s \in S} |V_k(s) - V_{k-1}(s)| < \frac{(1-\gamma)\epsilon}{2\gamma}. \tag{2.15}$$

This is an example of an *anytime* algorithm, where at any timestep, an approximate solution can be extracted.

Theoretically, it would be nice to be able to always solve for the optimal policy. In practice, this rarely occurs, since for most realistic domains, solving for the optimal policy is not tractable. Thus, much of the research in the planning field has focused on finding approximate solutions. In the MDP framework, a popular method for approximating the value function with large or continuous state or action spaces is through the use of function approximations [53, 54].

2.2 Partially Observable Markov Decision Processes

2.2.1 Description

Partially Observable Markov Decision Processes (POMDPs) provide a flexible decision making framework for partially observable domains [3, 12, 25, 30, 33, 51]. An environment which is partially observable means that the agent does not have direct access to its own state, but must infer the situation based on the (possibly noisy) observations it receives. POMDPs are well suited for probabilistic domains where decision making is required in the presence of sensor uncertainty, such as dialogue management or robot control. The POMDP model is the natural partially observable extension to the MDP model. We can formally define a POMDP as a tuple $(S, A, Z, T, O, R, \gamma)$. The characteristics S, A, T, R, and γ are inherited from the MDP model, where S is the set of states, A a set of actions, T is the transition function, R is the reward function, and γ is the discount factor. We refer to the MDP (S, A, T, R) as the *underlying MDP* of the POMDP. We now define the remaining terms Z and O:

Observations $Z = \{z_o, z_1, \ldots, z_L\}$ is the finite set of all possible observations the agent could receive from the environment. An observation is a piece of (possibly noisy) information about the environment. Only one observation z_t is seen at each timestep t, but multiple observation streams can be composed (via a crossproduct) into a single set of observations Z. It is possible to define POMDPs with continuous observations but for this work we examine the finite observation space case.

Observation Function We define the observation function O as

$$O(s, a, z) = P(z_{t+1} = z | s_t = s, a_t = a) \forall t. \tag{2.16}$$

This function defines the probability of observing z when executing action a in state s. We restrict $\sum_{z \in Z} O(s, a, z) = 1 \; \forall s, a$ to ensure that it is a probability distribution.

2.2.2 History and Belief States

In a partially observable environment, the agent cannot access the current state s_t directly. However, the agent receives observations at each step, which provides indirect access to the state. In general, these observations are insufficient to guarantee state knowledge.

The most obvious way to keep track of what state the agent is in is by keeping a *history* of the agent's actions and observations. We define a history h_t as

$$h_t = \{a_0, z_1, a_1, z_2, ..., a_{t-1}, z_t\}. \tag{2.17}$$

However, storing the history is too unwieldy to be practical for many tasks. Instead of maintaining the history, we can represent it by a *belief state* b_t, which is the probability distribution over states, given a history [3]. The belief state is a sufficient statistic for the history [47]. We can define b_t as

$$b_t(s) = P(s_t = s | h_t, b_0), \tag{2.18}$$

where b_0 is the initial belief state.

The advantage of using the belief state is that it offers a compact way of encapsulating the agent's history. Using the belief, we do not need to keep track of histories. We refer to the set of possible beliefs as ΔS. Note that ΔS is simply the simplex for the state space. Therefore, ΔS has dimension $(|S| - 1)$, and is a continuous space, even when the underlying MDP state space is finite.

Belief points also have the advantage of being easy to compute if the previous belief point is known, assuming we have the previous action and observation. We denote τ as the function which transitions from a previous belief state b_{t-1} to a new belief state b_t:

$$\begin{aligned} b_t(s') &= \tau(b_{t-1}, a_{t-1}, z_t), && (2.19) \\ &= \frac{1}{P(z_t | b_{t-1}, a_{t-1})} O(s', a_{t-1}, z_t) \sum_{s \in S} T(s, a_{t-1}, s') b_{t-1}(s). && (2.20) \end{aligned}$$

The next belief state at s' is simply the probability of transitioning to s' based on the previous belief, multiplied by the probability of the observation z_t occurring for that belief s', and normalized by the probability of receiving the observation z_t,

$P(z_t|b_{t-1}, a_{t-1})$. This factor can be computed as

$$P(z_t|b_{t-1}, a_{t-1}) = \sum_{s' \in S} O(s', a_{t-1}, z_t) \sum_{s \in S} T(s, a_{t-1}, s') b_{t-1}(s). \tag{2.21}$$

This belief update process is the same as the standard Bayes filter [23].

2.2.3 Policies and Value Functions

Much like in the MDP case, we can define a policy which specifies how an agent will act when operating within a POMDP environment. We define a policy π for a POMDP as

$$\pi(b) \rightarrow a. \tag{2.22}$$

Note that this is the same as the policy for the MDP, but we choose the action based on the belief state b, instead of based on the state s.

For the POMDP policy, the belief acts as the state in the MDP case. In fact, the POMDP itself can be represented by a MDP we call the *Belief MDP* (MDP_{Belief}) [25]. Since the belief transition function τ is a sufficient statistic, we can formulate $MDP_{Belief} = \{\Delta S, A, \tau, R_{belief}\}$, where the state space is the belief space ΔS of the POMDP, the action space A remains the same as the action space for the POMDP, the transition function is the belief transition function τ and the reward function R_{belief} is the reward for the beliefs:

$$R_{belief}(b, a) = \sum_{s \in S} b(s) R(s, a). \tag{2.23}$$

Since there always exists an optimal deterministic policy $\pi^*(b)$ for MDP_{Belief}, and since the belief MDP is functionally equivalent to the POMDP, therefore there always exists an optimal deterministic policy for a finite POMDP under a finite-horizon.

Using the notion of the Belief MDP and the standard value function update equation from Equation 2.12, we can create the value function update equation for the POMDP:

$$V_{k+1}(b) = \max_{a \in A} \left[\sum_{s \in S} b(s) R(s,a) + \gamma \sum_{z \in Z} P(z|b,a) V_k(\tau(b,a,z)) \right], \tag{2.24}$$

and similarly, our POMDP policy is defined:

$$\pi_{k+1}(b) = \operatorname*{argmax}_{a \in A} \left[\sum_{s \in S} b(s) R(s,a) + \gamma \sum_{z \in Z} P(z|b,a) V_k(\tau(b,a,z)) \right]. \tag{2.25}$$

Although a POMDP can be easily reduced to a MDP, planning for a POMDP is still a more difficult problem than MDP planning. The most obvious reason is that the size of the policy space is generally much larger. In a problem with $|S|$ states, ΔS is a $|S| - 1$ dimensional continuous space. This is referred to as the *curse of dimensionality*. Another issue is related to the size of the space of reachable belief points, where a reachable belief point is one which can be arrived at by the agent starting in the initial belief state b_0 and following some policy. This space is affected by the size of the action and observation spaces, as well as the length of the exploration horizon. This is referred to as the *curse of history*.

2.2.4 Planning

POMDP solving, or POMDP planning, is the task of computing a policy (optimal or approximate), given a full POMDP model. This is orthogonal to the problem of finding a POMDP model of an environment, which is a problem often addressed using reinforcement learning [7,43].

As we have seen, there are large computational challenges for POMDP solving. In this section, we will briefly review the literature on POMDP planning techniques, looking at both exact and approximate planning techniques.

Exact Planning

There have been many different algorithms for exact POMDP solving [14,16, 27,51,58,59]. Sondik's Enumeration [51] algorithm was not the first proposed algorithm, but is one of the most straightforward to describe. This algorithm is the POMDP analogue of the standard MDP value iteration algorithm seen in Section 2.1.2. Much like regular value iteration, the Enumeration algorithm performs iterations of dynamic programming to give successively better approximations to the value function.

Representing the value function for a POMDP is more complex than for an MDP, since the belief space is continuous, as well as being of much higher dimension as compared to the state space. Therefore, we cannot simply keep a tabular array to keep track of the value of each belief. Sondik showed that for a finite planning horizon, the optimal value function is a piecewise linear, convex function of the continuous belief space [47] [51]. Therefore, we can represent the value function for any finite planning horizon t as a set of vectors $\Gamma_t = \{\alpha_0, \alpha_1, \ldots \alpha_{|\Gamma_t|}\}$. These α-vectors are hyperplanes existing in belief space. α-vectors are associated with a specific action, and they represent an estimated value over the belief space, where the value for an

α-vector is associated with the total return the agent would receive when executing the associated action at that belief.

We can compute the value an α-vector associated with a belief via an inner product:

$$V^\alpha(b) = \sum_{s\in S} \alpha(s)b(s). \tag{2.26}$$

The α-vectors taken as a set, provide a full value function:

$$V_t(b) = \max_{\alpha\in\Gamma_t} \sum_{s\in S} \alpha(s)b(s), \tag{2.27}$$

where Γ_t is the set of α-vectors at timestep t. An example value function is shown in Figure 2–1.

We say an α-vector α is dominated if:

$$\forall b \in \Delta S\ \exists \alpha' \in \Gamma\ s.t. \sum_{s\in S} \alpha'(s)b(s) \geq \sum_{s\in S} \alpha(s)b(s). \tag{2.28}$$

A dominated α-vector is never the best α-vector at any point in the belief space. Since $V(s)$ represents a lower bound to the true value, a dominated α-vector can be safely removed (pruned) without changing the value function. We also say that an α-vector α_1 dominates α_2 iff $\alpha_1(s) \geq \alpha_2(s)\ \forall s$.

The Enumeration algorithm dynamically builds a set of α-vectors, where our set of α-vectors for each planning horizon t is denoted Γ_t. We build an intermediate set of alpha vectors which are used in the following computation. This initial set of α-vectors corresponds to the immediately received rewards, since we only consider a

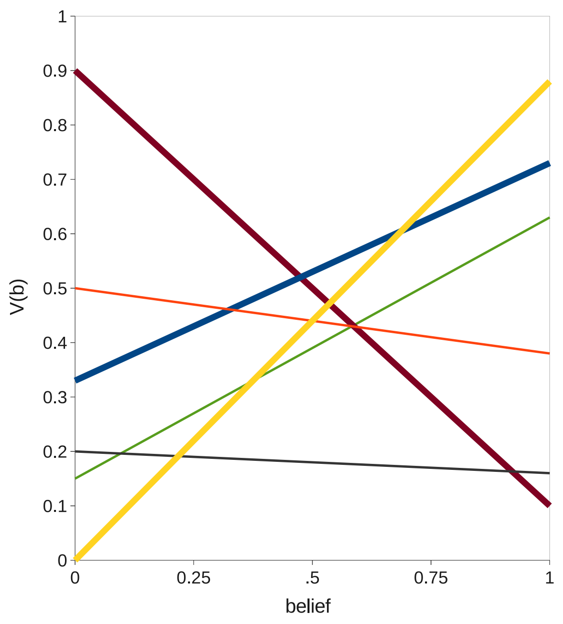

Figure 2–1: A sample value function, shown here with six α-vectors. This POMDP has two states, so the belief is a continuous value between 0 and 1. We note that only three α-vectors are being used (denoted by the thicker lines). All other α-vectors are dominated by some other vector at some belief point.

single step. These are separated per action:

$$\Gamma_1^a = \{\alpha^a | \alpha^a(s) = R(s,a) \forall s\}. \tag{2.29}$$

We use these to create our initial set of α-vectors, which we denote as Γ_1:

$$\Gamma_1 = \bigcup_{a \in A} \Gamma_1^a. \tag{2.30}$$

Now we build Γ_t based on Γ_{t-1}. We combine the immediate reward step with the plans of length $(t-1)$. First, we build the sets $\{\Gamma_t^{a,z}\} \forall a, z$. Each $\Gamma_t^{a,z}$ is a set of α-vectors associated with taking action a and receiving observation z, and afterwards following the plans of length $(t-1)$.

For a single action a and observation z, we build

$$\Gamma_t^{a,z} = \Big\{ \alpha_i^{a,z} | \alpha_i^{a,z}(s) = \gamma \sum_{s' \in S} T(s, a, s') O(s', a, z) \alpha_i'(s'), \alpha_i' \in \Gamma_{t-1} \Big\}. \tag{2.31}$$

We combine these with the one-step reward vectors using the cross-sum[1] :

$$\Gamma_t^a = \Gamma_1^a \oplus \Gamma_1^{a,z_1} \oplus \Gamma_1^{a,z_2} \oplus \ldots \oplus \Gamma_1^{a,z_{|Z|}}, \tag{2.32}$$

and finally combine all Γ_t^a sets:

$$\Gamma_t = \bigcup_{a \in A} \Gamma_t^a. \tag{2.33}$$

This algorithm can solve POMDPs with finite state and action space given the choice of planning horizon. The actual value function V_t can be extracted from the set of α-vectors as described in Equation 2.27.

There have been many other exact planning algorithms [14, 16, 27, 51, 58, 59]. The Witness [27] algorithm uses a different criterion for when to create α-vectors. It considers regions where current α-vectors are not optimal, and then adds in the

[1] The cross-sum ($\oplus$) operator is defined: for sets $A = \{a_1, a_2, \ldots, a_{|A|}\}$, $B = \{b_1, b_2, \ldots, b_{|B|}\}$, $A \oplus B = \{a_1 + b_1, a_1 + b_2, \ldots, a_1 + b_{|B|}, a_2 + b_1, \ldots, a_{|A|} + b_{|B|}\}$

optimal vector. More recent algorithms have focused on removing dominated α-vectors, since in the exact methods, large numbers of α-vectors must be created. Using heuristics to select belief points for the updates has also been explored [59].

The exact methods have become increasingly sophisticated. Unfortunately, exact planning methods still carry a very high computational cost. For a single planning iteration at a horizon of t, the computation cost is $O(|A||Z||S|^2|\Gamma_{t-1}|^{|Z|})$. Even with clever pruning of α-vectors, the size of Γ remains exponential in Z. For practical domains, it is essential that we consider approximate techniques.

Approximate Planning

There are a wide variety of existing approaches for approximating a value function for a POMDP. In this section, we describe some of the most common methods.

Heuristic Approaches. Several methods have been proposed that use heuristics based on the underlying MDP, including Most Likely State (MLS) [35], $QMDP$ [28] and Fast Informed Bound (FIB) [21]. As we have discussed, MDP planning is much simpler than solving the full POMDP. While following the optimal MDP policy will be suboptimal in a partially observable domain, it can, nevertheless, be a useful heuristic in many cases. We solve for Q_{MDP} as described in Equation 2.10, where our MDP is the underlying MDP of the POMDP, specifically $MDP = (S, A, T, R)$. The MLS heuristic creates a policy simply by choosing the action for the state that is most likely in the belief:

$$\pi_{MLS}(b) = \operatorname*{argmax}_{a \in A} Q_{MDP}(\operatorname*{argmax}_{s \in S} b(s), a). \tag{2.34}$$

This technique can perform well when there is generally a strong peak in the belief state, which is common in robotic applications with powerful sensors. However, MLS will perform poorly in any situation where there is a large amount of uncertainty, and it would never select information gathering actions. We note that this is an upper bound for the true value function as it is optimistic in assuming we have full observability. A slightly more complex heuristic is $QMDP$. We define the heuristic as :

$$\pi_{QMDP}(b) = \operatorname*{argmax}_{a \in A} \sum_{s \in S} b(s) Q_{MDP}(s, a). \tag{2.35}$$

This creates a single α-vector for each action, i.e. $\alpha^a(s) = Q_{MDP}(s, a) \forall a$. This is still an upper bound, but tighter than the one proposed by V_{MLS}. The advantage of $QMDP$ over MLS is that it takes into account the uncertainty at the current step. However, it still has no ability to do long term information gathering, since all future steps assume full observability.

Both of these methods rely on the fully observable underlying MDP policy, without taking into account any uncertainty. Hauskrecht's FIB heuristic integrates the observation weights into the update step (here we represent the Q function as a set of α-vectors).

$$\alpha^a_{t+1} = R(s, a) + \gamma \sum_{z \in Z} \max_{\alpha_t \in \Gamma_t} \sum_{s' \in S} O(s', a, z) T(s, a, s') \alpha_t(s'), \tag{2.36}$$

and we define the value function as in Equation 2.27.

The intuition behind the FIB heuristic is that it selects the optimal action weighted by the expected observation. The value function remains an upper bound as before, but is guaranteed to be below the $QMDP$ value function (i.e. $V^*_{POMDP}(b) \leq V_{FIB}(b) \leq V_{QMDP}(b)\ \forall b$).

Policy Search. Most of the techniques we have discussed so far create a policy by iteratively building a value function. Another approach is to directly optimize the policy. We call these approaches policy search algorithms. A policy search algorithm for POMDPs depends on a) how to represent the policy and b) how to learn the policy.

Hansen [19] proposed using a finite state machine (FSM) to represent the policy. In this formulation, each node in the policy graph is associated with the action that the agent should take when in that node. Transitions between the policy nodes are associated with observations, so when the agent receives an observation, it moves to the next node in the FSM. Note that as described, this results in a deterministic policy although a stochastic policy could be formed as well. It is straightforward to extract a value function from this policy representation. Policy iteration is done by directly modifying the FSM, i.e. by adding, modifying and removing nodes. Other techniques are possible, including branch-and-bound methods [32] and gradient ascent [6, 8, 32, 34, 57]. Bounded Policy Iteration [41] uses gradient ascent to search for policies while restricting the FSM to a fixed size. Once encountering a local optima, the size of the controller is increased by one (or a few) nodes to escape the optima.

Policy search techniques apply more easily to POMDPs with continuous parameters (such as continuous state or action spaces) than do value function approaches. However, selecting the class of policies to be explored can be difficult. As well, local optima are more of an issue with policy search. However, policy search techniques have been applied to real-world domains [5].

Grid-based. Grid-based value function algorithms [11,20,29,60] approximate the value function over a subset of the true belief space. These belief points are selected in a grid pattern, which we denote $G = \{b_1^G, b_2^G \dots b_{|G|}^G\}$. The goal of arranging the belief points in a grid is to give a wide coverage over the belief space. For belief points not covered by the grid itself, an interpolation of the values of the nearby beliefs on the grid is used:

$$V(b) = \sum_{i=0}^{|G|} \lambda(i) V(b_i^G). \tag{2.37}$$

where $\lambda(i) \geq 0\ \forall i$, and $\sum_{i=0}^{|G|} \lambda(i) = 1$. To compute the values $V(b_i^G)$, we use standard value iteration for POMDPs, as in Equation 2.24.

The convex combination function $\lambda(i)$ can be found by solving a linear program [60]. However, solving the system exactly can be very computationally expensive. Since any convex combination will give an upper bound to the value function, approximate solutions to compute $\lambda(i)$ have been proposed.

The grid-based approaches vary in how they pick their grid points G, and how the interpolation function $\lambda(i)$ is computed. The different approaches therefore will provide different optimality guarantees. One major drawback to the grid-based approaches is the difficulty in forming effective policies when important beliefs are dense in specific areas of the belief space. As well, since the grid-based approaches tend to not consider reachability of the belief points, the representation does not focus computation on the parts of the belief space which are most useful (i.e. the reachable subspace).

Online Approaches. All algorithms already discussed are types of *offline methods*. This means there is a clear separation between the POMDP solving and the execution of the resulting policy. In offline solvers, the POMDP is solved ahead of time, producing a fixed policy. This fixed policy is subsequently used in execution. *Online methods* remove the separation between solving and execution, by interleaving POMDP updates while the agent moves through the environment. An advantage in using online approaches is that the solver only needs to plan based on the current belief state of the agent, instead of solving for the entire belief space. Furthermore, since the online planning occurs at every timestep, generalization to other belief states is not nearly as important, since subsequent belief states undergo replanning regardless.

Online approaches accomplish this by building a tree of belief nodes with the current belief of the agent as the root, and using the normal belief transition $\tau(b, a, z)$ (Equation 2.19) to generate subsequent nodes. The tree is built with action decision nodes between layers of belief nodes. The size of the tree (i.e. the planning horizon) is generally selected based on the real-time constraints of the agent. If the agent has more time for real-time planning, then the size of the planning horizon can be extended. Once the tree is built, the fringe belief nodes values are seeded with the results from an offline solver, or some other heuristic. The belief point values are propagated upwards through the tree using the standard Bellman Equations (see Equation 2.24).

We have described the general guideline for an online solver. There are a variety of published techniques, which vary in how the belief points are selected, or, viewed another way, how they are pruned from exploration. These methods include using heuristics [42, 55], Branch-And-Bound pruning [36] and Monte-Carlo sampling [31]. See [44] for a recent survey on online planning techniques.

Using an online solver has been shown to be very efficient for POMDPs with large state or action spaces. Online solvers do have the restriction that all computation must be done while the agent is executing within the environment, which may be a minimal amount of time, depending on hardware and real-time constraints.

We note that the use of an online planner is completely orthogonal to the use of an offline planner. While executing actions selected by online planning technique, the leaf belief nodes in the planning tree must be initialized with the result of an offline solver. It is clear that the quality of the offline value function approximation will affect the final performance of the online planner, especially in the case of tasks with limited real-time computation. The focus of the latter chapters of this thesis is on offline solvers. However, if the real-time constraints of the environment allow an online solver to be used, then applying online backups to the offline value function is recommended.

Point-Based. A final category of offline POMDP approximation techniques is point-based methods, which have become very popular due to their ability to solve large problems much more efficiently than exact methods. The collecting and updating algorithms explored in this paper are associated with the collection/updating methods of the point-based algorithms in the literature. Therefore we need to describe the current point-based methods in detail. We describe these point-based algorithms in the next chapter.

CHAPTER 3

Point-Based POMDP Solvers

As we have seen, there are a wide variety of POMDP planning techniques, both exact and approximate. Research in POMDP planning is ongoing, however, due to the steep computation complexity of the framework. We have discussed two of the key challenges of POMDP planning: the curse of dimensionality, referring to the exponentially increasing belief space, and the curse of history, referring to the exponentially increasing size of the required planning space as the planning horizon grows. Point-based methods tackle these challenges by careful selection of belief points, by choosing points that will support a more accurate value function for the POMDP. These methods also attempt to generalize from these individual points, by using the methods developed for exact POMDP planning to give good value function approximations for points in the belief space that were not chosen.

Point-based planners draw from the grid-based methods in how they approximate via choosing only a subset of belief nodes over which to iterate. These planners are also influenced by the heuristic-based methods we have discussed.

There have been a variety of point-based POMDP planners proposed in the literature. These planners differ in several places, including the methods for collecting points, updating points, and other optimizations. The focus in this thesis is to examine what are the best methods to use for point-based solvers.

In the next section, we will review the core algorithmic component found in point-based POMDP solvers. The rest of the chapter will describe several of the well-known point-based solvers from the literature. The goal is to describe the history of the algorithms, and to give a high level description of the details of their subcomponents. Note that we do not compare these algorithms directly in the latter part of this thesis; rather, our empirical work compares the core components of these algorithms, in an attempt to better understand their effectiveness.

3.1 Introduction to Point-Based POMDP Solvers

In the exact planning methods we have discussed, the goal is to create a new set of α-vectors such that the α-vector for each belief point in the simplex is updated. Instead of looking at the whole space as before, let us focus on a single belief point b. We define the *backup* operation on a belief point b as:

$$\begin{aligned} backup(b) &= \alpha_t^b & (3.1) \\ &= \operatorname*{argmax}_{\alpha \in \Gamma_t} \sum_{s \in S} b(s)\alpha(s). & (3.2) \end{aligned}$$

Therefore the α-vector created by the $backup(b)$ operation is the α-vector in the (unknown) value function V_t which gives the maximum value for the belief. We think of this as “updating” or “backing up” the belief point since the operation computes the new α-vector by back-propagating all current α-vectors one step and combining it with the one-step reward. This is an instance of dynamic programming, similar in principle to the original Bellman updates. We stress that $backup(b)$ is an operation, since it computes a new α-vector, rather than a simple assignment.

Point-based solvers, instead of ensuring that the value function is updated over the entire belief simplex ΔS, only apply the backup operation over some specified finite subset of belief points B such that $B \subset \Delta S$. Here we describe how a point-based solver would create the α-vectors for the next value function.

First, we need to build our intermediate sets Γ_1^a (the immediate reward vectors) and $\Gamma_t^{a,z}$, our plans of length $(t-1)$ based on our previous set of α-vectors Γ_{t-1}:

$$\Gamma_1^a = \{\alpha^a | \alpha^a(s) = R(s,a) \forall s\}, \tag{3.3}$$

$$\Gamma_t^{a,z} = \Big\{\alpha_i^{a,z} | \alpha_i^{a,z}(s) = \gamma \sum_{s' \in S} T(s,a,s') O(s',a,z) \alpha_i'(s'), \alpha_i' \in \Gamma_{t-1}\Big\}. \tag{3.4}$$

The operations to compute these sets are the same as in the exact methods. We now define the set of vectors Γ_b^a:

$$\Gamma_{t,b}^a = \Gamma_1^a + \sum_{z \in Z} \operatorname*{argmax}_{\alpha \in \Gamma_t^{a,z}} \left[\sum_{s \in S} \alpha(s) b(s)\right]. \tag{3.5}$$

Whereas in the exact value update, we combined all possible combinations of observations and previous plans, here we are simply summing over observations on the best previous vector for the specific belief state.

Finally, our *backup* operation becomes:

$$backup(b) = \operatorname*{argmax}_{\alpha \in \Gamma_{t,b}^a} \sum_{s \in S} b(s) \alpha(s). \tag{3.6}$$

The backup operation creates the set $\Gamma_{t,b}^a$ for the specified belief point. It then picks the α-vector in that set that maximizes the value at the belief point b. Note that in the point-based case, we only build $\Gamma^a_{t,b}$ when required, instead of computing it for all possible belief points under a fixed horizon t, as in the exact planning methods. As well, since the $\Gamma^{a,z}_t$ sets are used in multiple $\Gamma^a_{t,b}$ computations, these sets can be stored to save computation time. The full *backup* operation runs in $O(|S|^2|A||Z||\Gamma_{t-1}|)$ time.

The key assumption of point-based planners is that the finite set of belief points that are updated will generalize well enough to cover the reachable subspace of the belief space. The intuition behind this assumption is that the set of important, reachable beliefs are not uniformly distributed in ΔS, but form some sort of correlated subspace. If this is the case, then the set of points selected should generalize well, as long as the set B is collected carefully.

All point-based methods share this *backup*(b) operation. The algorithms differ mainly in three respects. First, these algorithms differ in the method used to select the new belief points. Second, they differ in which belief points are updated in each backup iteration, as well as the ordering of the updates. Third, the various point-based algorithms have different minor optimizations, such as pruning techniques or better initial value functions. In general, these algorithms interleave steps of belief point collection and belief point updates. An outline of a generic point-based algorithm is shown in Algorithm 1. The $COLLECT$ method selects the new belief points, while the $UPDATE$ method determines which belief points will be updated, and in which order.

The algorithm alternates between belief point collection and iterations of belief point updates, and terminates at some stopping condition. The stopping condition

Algorithm 1 Point-based POMDP Solver

```
$t \leftarrow 0$
Initialize $V_0$
Initialize $B$
**while** stopping condition not met **do**
    $B_{new} \leftarrow COLLECT(V_t, B, N)$
    **for** U iterations **do**
        $V_{t+1} \leftarrow UPDATE(V_t, B, B_{new})$
        $t \leftarrow t + 1$
    **end for**
    $B \leftarrow B \cup B_{new}$
**end while**
```

could be time (especially for a time-constrained planner) or a fixed number of iterations. Some point-based methods provide a guarantee on the quality of the resulting policy, where the stopping condition is specified in relation to the desired quality of the policy. Since at any step the current value function V_t could be used to generate a policy, point-based algorithms have the advantage of being anytime algorithms. Each iteration of belief point collection selects N belief points. We also see that the algorithm runs U iterations of belief point updates.

There have been a few proposed ways to initialize the value function V_0. In general, point-based methods use a lower bound initialization for V_0 over the entire belief simplex ΔS. An advantage of this approach is that with V initialized as a lower bound, it will remain a lower bound for the entire computation, which makes it easier to detect convergence.

A common method to initialize V_0 is to add a single α-vector such that:

$$\alpha(s) = \frac{\max_{a \in A} \left[\min_{s \in S} R(s, a) \right]}{1 - \gamma} \forall s \in S. \tag{3.7}$$

This gives us an α-vector associated with the action which gives us the best worst-case immediate reward. This provides a simple guaranteed lower bound.

We will now present some of the well-known point-based solvers in reference to this framework.

3.2 Point-Based Value Iteration (PBVI)

Point-based Value Iteration, published in 2003 [39], was an early point-based approximate solver. Pineau et al. introduced the notion of only applying the *backup* operation on a finite subset $B \subset \Delta S$ where all $b \in B$ are reachable from the initial belief b_0.

Collection

In PBVI, the belief set B is initialized with a single belief point, the initial belief point in the POMDP problem b_0. At each step, the set B is expanded by greedily choosing new reachable points that improve the worst-case density of the current belief set.

The collection algorithm iterates through belief points b currently in the set B. A set $\{b_{a_0}, b_{a_1}, \ldots\}$ is generated for each $b \in B$, where each b_{a_i} is created by executing action a_i in b, and sampling a random observation z from $O(s', a_i, z)$, where s' is a sampled next-state. b_{a_i} is then generated from $\tau(b, a, z)$ (Equation 2.19). Next, PBVI calculates the L_1 norm [1] between each belief in $\{b_{a_i}\}_i$ and every $b \in B$. Finally, it adds the successor belief point b_{max} which is the farthest (L_1 norm) from any point already in B:

$$b_{max} = \underset{b \in \{b_{a_i}\}_i}{\operatorname{argmax}} \left|\left|b_{a_i} - B\right|\right|_1 , \tag{3.8}$$

where we define $\left|\left|b_{a_i} - B\right|\right|_1$, the L_1 norm between a point and a set as:

$$\left|\left|b_{a_i} - B\right|\right|_1 = \max_{b \in B} \left|\left|b_{a_i} - b\right|\right|_1 . \tag{3.9}$$

The belief point b_{max} is computed for each b in the initial set B, so the belief space doubles at each iteration. The goal of using the L_1 norm metric to choose successor nodes is to expand the belief set towards new beliefs which are far away in belief space. Note that since all beliefs added are successor beliefs, all points in B are guaranteed to be reachable.

Backup

PBVI starts each iteration with a value function empty of α-vectors. PBVI then simply iterates through all belief points $b \in B$, and adds $\alpha = \mathit{backup}(b)$ to the value function, as long as that α-vector doesn't already exist in the value function. This results in a value function with at most $|B|$ α-vectors.

[1] The L_1 norm is a distance metric between two points where the distance is the sum of the absolute differences of their coordinates, i.e. $\left|\left|b - b'\right|\right|_1 = \sum_{s \in S} |b(s) - b'(s)|$. This is also known as the Manhattan distance.

Parameters and Other Modifications

The stock PBVI algorithm alternates between belief collection and a parameter T iterations of backups. PBVI does not require pruning, since at each step the set of α-vectors Γ_t is initialized with no α-vectors. The built-in pruning step makes the algorithm more efficient since a reduced size of the set of α-vectors Γ leads to a faster $backup(b)$ operation.

PBVI also provides convergence results, which bound the error in the value function based on the density of B within ΔS [39].

3.3 Perseus

Spaan and Vlassis presented the Perseus algorithm in 2004 [52]. Perseus introduces a method to reduce the number of belief updates required at each iteration, while still improving the value function for a large number of beliefs. With less belief point updates per step, there is room for many more iterations of belief point updates, for a given time constraint.

Collection

Perseus builds its set of belief points through a random exploration of the environment. Unlike many of the other algorithms, which iterate between steps of belief point collection and backups, Perseus builds a fixed set of belief points B at the start. B is collected by sampling random trajectories starting at the initial belief b_0, and selecting a random action a uniformly at each step. This algorithm still fits the framework described in Algorithm 1, as we initialize B_0 to this set of random trajectory belief points, and each expansion stage adds zero new beliefs.

Backup

The primary goal in Perseus' backup stage is to improve the value function approximation with the least belief point backups required. The backup stage guarantees that the approximation is improved for all belief points in B, while only running backups on a randomized subset of B.

At each step in the iteration, the next value function V_{n+1} is initialized without any α-vectors (i.e. $\Gamma_{n+1} = \emptyset$), and we initialize a list of unimproved belief points $\tilde{B}$ to the full set B. As long as $\tilde{B}$ is not empty, we randomly sample a belief point b from $\tilde{B}$ and $\alpha = \mathit{backup}(b)$ is computed. If this α-vector provides a better value than the value from the previous iteration (i.e. $\sum_{s \in S} b(s)\alpha(s) \geq V_n(b)$), then this new α-vector is added to V_{n+1}, otherwise the previous best vector $\operatorname{argmax}_{\alpha \in \Gamma_n} \sum_{s \in S} b(s)\alpha(s)$ is added to V_{n+1} to ensure termination of the iteration. Then all belief points which have been improved or maintained by this new vector ($V_{n+1}(b) \geq V_n(b)$) are removed from $\tilde{B}$. The process repeats until $\tilde{B}$ is empty and therefore all belief points have been either improved or at least maintained. When the set of unimproved belief $\tilde{B}$ is empty, then $\tilde{B}$ is reinitialized to B and the process repeats until some termination criterion is reached.

The advantage of this method for choosing points to update is that it is guaranteed to improve or at least maintain all beliefs at each iteration, just as a full backup episode would. In general, it will require many fewer point-based backups, since a single backup can improve the value for many belief points in B, however, in the worst case, it will still require $|B|$ backups. A full backup might give a better resulting value function overall, since the resulting value for a backup at a specific belief b will in general be higher than the value for b under an α-vector created by some other belief b'.

Parameters and Other Modifications

The Perseus update procedure has a built-in pruning mechanism, as the only α-vectors that are added to the next step's value function are the newly backed up α-vectors, or α-vectors that were found to support a belief point in the previous value function. This means that each new value function will only have α-vectors which support at least a single belief point $b \in B$.

A parameter, $|B|$, must be specified for Perseus, giving the size of the set of beliefs collected. In the paper, they vary the size of $|B|$ depending on domain characteristics, mainly the size of the state space.

3.4 Heuristic Search Value Iteration (HSVI)

Heuristic Search Value Iteration was published in 2004 [48] and was updated in 2005 [49]. HSVI uses heuristics based on upper and lower bounds of the value function to collect beliefs. The algorithm was introduced as an improvement over the state of the art in POMDP solving, as using the heuristics to guide the search in belief space would lead to critical beliefs faster than PBVI. In this section we will cover the updated version of HSVI [49].

Collection

As in general point-based methods, HSVI iteratively builds a value function V, represented as a set of α-vectors Γ. Recall that V represents a lower bound to the optimal value function V^*. HSVI additionally constructs an upper bound $\overline{V}$. We refer to the pair of bounds as $\hat{V}(b) = [V(b), \overline{V}(b)]$.

HSVI defines the *width* function:

$$width(\hat{V(b)}) = \overline{V}(b) - V(b), \tag{3.10}$$

which is the gap between the bounds at the belief point b.

HSVI uses a *point set* $\Upsilon_{\overline{V}}$ to represent the bound $\overline{V}$. The point set is a set of belief-value pairs $(b_i, \overline{v}_i)$, and updates to $\overline{V}$ are done by adding the belief point b and upper bound value at b to $\Upsilon_{\overline{V}}$. The value at any given belief b is calculated by projecting it on the convex hull of the set. This is calculated in the earlier version of HSVI by solving the linear equations, but the newer version calculates the upper bound of the value function via an approximation introduced by Hauskrecht [21].

For the belief collection process, HSVI starts at the initial belief point b_0. At each step in the exploration, the algorithm picks an action a^* and an observation z^* based on heuristics. The action is picked such that $a^* = \text{argmax}_{a \in A} Q^{\overline{V}}(b, a)$, where

$$Q^{\overline{V}}(b, a) = \sum_{s \in S} \Big[R(s, a)b(s)\Big] + \gamma \sum_{z \in Z} \Big[P(z|b, a)\overline{V}(\tau(b, a, z))\Big]. \tag{3.11}$$

This action selection is known as the $IE - MAX$ heuristic [24]. The action a^* is chosen based on the upper bound, since the upper bound value for an action can only decrease on an update. An action based on the maximum upper bound value can be either truly the optimal action, or will have its upper bound estimate decreased on an update. If the action was shown to be suboptimal, the upper bound for that action will decrease, and then a new action will be the optimal upper bound action. Therefore, picking the maximum upper bound action will result in the most accurate upper bound estimate. HSVI defines the *excess* function:

$$excess(b,t) = width(\hat{V}(b)) - \epsilon\gamma^{-t}, \tag{3.12}$$

where ϵ is an input parameter to the algorithm bounding the quality of the solution. The excess is the uncertainty about the belief (expressed as the width between the bounds), subtracted by a parameter which grows as the depth of the search grows. HSVI's collection method picks the next observation z^* based on which observation z contributes the most uncertainty to the current belief state:

$$z^* = \operatorname*{argmax}_{z \in Z} \Big[P(z|b,a^*) excess(\tau(b,a^*,z), t+1) \Big]. \tag{3.13}$$

The goal for these heuristics is to guide the search towards beliefs which contribute the most to the difference between the upper and lower bounds. The search terminates once we arrive at a belief b such that $excess(b,t) < 0$. The intuition behind this bound is that the search should continue if there is still a large width in the successor beliefs, but to terminate if the search is so deep that the beliefs are not as relevant.

The belief terminates at some stopping condition, either reaching a goal state, some fixed number of iterations, or some other condition.

Backup

After HSVI has collected a trace of belief points, it proceeds to update the bounds for each belief, one at a time. The belief points are processed in reverse order, which should give a better resulting value function due to how the *backup* operation uses the α-vectors of next-state beliefs. At each new belief in the search, both the upper and lower bounds are updated. The lower bound $V(b)$ is updated by adding $\alpha = \mathit{backup}(b)$ to Γ. The upper bound $\overline{V}(b)$ is updated by adding the belief-value pair $(b, \max_{a \in A} Q^{\overline{V}}(b, a))$ to the point set $\Upsilon_{\overline{V}}$.

In HSVI, only the newest beliefs are updated at each step, instead of the entire set as in PBVI. The advantage is that less time is spent updating beliefs which, since they have been previously updated, are in theory less likely to produce better bounds. The tradeoff is that more time (proportionally) must be spent collecting new points instead of updating the bounds.

Parameters and Other Modifications

Whereas many POMDP solvers use a simple worst-case α-vector as a lower bound as seen in Equation 3.7, HSVI uses a *blind policy* to initialize the lower bound, first proposed by Hauskrecht [20]. An α-vector α^a is maintained for each action, associated with the policy of "always take action a". α_0^a is initialized to the worst case α-vector as described in Equation 3.7. The vector is updated with the Bellman update:

$$\alpha_{t+1}^a(s) = R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a)\alpha_t^a(s'). \tag{3.14}$$

Once the iteration is finished, the value function is initialized with the set α_t^a. The resulting bound is guaranteed to be at least as tight as the single α-vector bound from Equation 3.7, since the value function never decreases. The tradeoff is minor since the computation only must be done once and is quite fast relative to the POMDP planning.

HSVI uses the Fast Informed Bound (FIB) (Section 2.2.4) to initialize the upper bound $\overline{V}$. The initial α-vectors for the FIB update (α_0^a) are initialized with the solution to the underlying MDP.

3.5 Point-Based Error Minimization Algorithm (PEMA)

The Point-Based Error Minimization Algorithm (PEMA), introduced in 2005, proposed a theoretically-motivated approach to selecting belief points, via selecting points based on which point minimizes the bound on the error of the value function [38].

Collection

The goal of the collection method in PEMA is to select reachable belief points which have the greatest difference between their current value and the maximum value that it could reach given an optimal policy for the completed POMDP. The idea is that these points with large errors have the most room for improvement in the lower bound approximation for the value function, and updating these belief points first will lead to a faster solution.

Pineau et al. [39] showed that for a given planning horizon t and a fixed set of belief points B, the error over the value iteration is bounded by

$$\left|\left|V_t^* - V_t^B\right|\right|_\infty \leq \frac{(R_{max} - R_{min}) \max_{b' \in \Delta S} \min_{b \in B} ||b - b'||_1}{(1-\gamma)^2}, \tag{3.15}$$

where V_t^B is the PBVI estimate value for a planning horizon t and belief set B, b' is the point in belief space where the approximated value function makes the worst error, and $b \in B$ is the closest (L_1 norm) point to b'. This is the overall error bound for the entire value function.

Pineau et al. [38] then use a similar argument to define the error bound for any belief point $b' \notin B$:

$$\epsilon(b') \leq \sum_{s \in S} \begin{cases} \left[\frac{R_{max}}{1-\gamma} - \alpha(s)\right] [b'(s) - b(s)] & \text{if } b'(s) \geq b(s) \\ \left[\frac{R_{min}}{1-\gamma} - \alpha(s)\right] [b'(s) - b(s)] & \text{if } b'(s) < b(s). \end{cases} \tag{3.16}$$

In this bound, b' is the target belief, $b \in B$ is the belief with the closest L_1 norm belief, and α is the α-vector in the current value function that is maximal for b. This bound provides a mechanism for finding (for some unexplored belief) the error between its current value and its value on an unknown α-vector.

PEMA also takes into account reachability concerns, by weighting the belief point by how likely the agent is to reach that belief. The algorithm defines a "potential error" heuristic $\overline{\epsilon}(b)$ for $b \in B$:

$$\overline{\epsilon}(b) = \max_{a \in A} \sum_{z \in Z} O(b, a, z) \epsilon(\tau(b, a, z)). \tag{3.17}$$

This heuristic takes into account the reachability of a belief as well as the error bound $\epsilon(b)$. Starting at an explored belief b, this checks every action a, and returns the highest next-belief error, which is weighted by the observation probability and calculated via the error of the corresponding next belief $\tau(b, a, z)$.

We now describe the belief collection process. A tree of belief points b is formed, with the initial belief b_0 as the root. Only children beliefs of nodes in the tree are considered for addition, which guarantees the reachability of all belief points. At each collection step, $\overline{\epsilon}(b)$ is calculated for all $b \in B$. The node with the highest $\overline{\epsilon}(b)$ is chosen, and the node b' is chosen such that:

$$b' = \tau(b, \left[\operatorname*{argmax}_{a \in A} \sum_{z \in Z} O(b, a, z)\epsilon(\tau(b, a, z))\right], \left[\operatorname*{argmax}_{z \in Z} O(b, a, z)\epsilon(\tau(b, a, z))\right]). \tag{3.18}$$

This new belief b' is the one which contributed most (in expectation) to the error at $\overline{\epsilon}(b)$. The belief b' is then added as a child node to the belief b.

Backup

The belief point update step in PEMA is modelled after the update step in PBVI. After the new belief points have been collected, a full backup is executed on all $b \in B$, incorporating at most $|B|$ α-vectors to the new value function.

3.6 Forward Search Value Iteration (FSVI)

Forward Search Value Iteration (FSVI) was introduced in 2007 [46]. This point-based algorithm uses an exploration heuristic based on using the underlying MDP to guide the search for new belief points.

Collection

Using the underlying MDP (i.e. $MDP = (S, A, T, R)$) as a heuristic to solve the overall POMDP has been explored in the literature and discussed in Section 2.2.4. The FSVI collection algorithm uses an MDP value function to guide the search through policy space.

The first step in the initialization of the overall algorithm is to build the state-action value function $Q(s, a)$ of the underlying MDP through standard value iteration (Equation 2.10). While a full solution needs to be computed, this step must only be done once. Each full step of state-action value iteration takes $O(|S|^2|A|^2)$ time, so the running time for a full solution with N iterations has running time $O(|S|^2|A|^2N)$. In practice, this step is small compared to the running time of the actual POMDP planning.

The collection algorithm takes the form of a joint search through belief and state space. FSVI starts every collection iteration at the initial belief b_0, and samples from the initial belief to start in an underlying state s_0.

At each step in the process, the FSVI collection algorithm maintains the pair (s, b). It picks the action a^* based on s and the action heuristic:

$$a^* = \operatorname*{argmax}_{a \in A} Q(s, a). \tag{3.19}$$

The action selected at each step is the optimal action for the state s in the underlying MDP. The intuition behind this approach is that it will guide the search for new beliefs towards more critical rewarding beliefs. The next state s' is sampled from $T(s, a, s')$. The observation z is sampled from $O(s, a, z)$. The stochastic observation selection guarantees that there will be variance in the trace through belief space, as well as keeping the beliefs weighted towards the likelihood of them being encountered. The next belief state b' is computed via the normal belief transition function $\tau(b, a, z)$. The process then continues with (s', b') becoming the new state-belief pair.

We continue in this fashion, growing a trace of points in belief space. This continues until we reach some stopping condition, such as if our current s is a terminal state in the POMDP, or some fixed number of collection steps.

The intuition behind this collection approach is to use the perfect-information MDP to guide the search through belief space towards beliefs which provide reward. An advantage of this method is that it is very efficient, since computing a^* only takes $O(a)$ time.

One issue with this heuristic is it may perform poorly in domains where it takes several steps to execute an information gathering action, such as in the Heaven-Hell problem [18]. The MDP-based heuristic will never purposefully lead it towards states which provide informative observations. This is because the MDP solution places no value on information gathering actions.

Backup

The backup operation in FSVI is similar to the operation in HSVI. Once the collection trace is complete, the belief points in the trace are backed up in reverse order (i.e. the final belief in the trace is backed up first). The resulting α-vectors are added to the current value function.

3.7 Successive Approximation of the Reachable Space under Optimal Policies (SARSOP)

SARSOP [26] was introduced in 2008 as an improvement over HSVI, by modifying the sampling approach to sample near $R^*(b_0)$, the subset of belief points reachable by b_0 which are explored under optimal action selection. Of course, knowing $R^*(b_0)$ exactly is impossible, since it requires the exact POMDP solution. The algorithm iteratively approximates $R^*(b_0)$ by first using the current estimate to update the value function bounds, $\underline{V}(b)$ and $\overline{V}(b)$, and then using the updated bounds to recompute $R^*(b_0)$. The SARSOP sampling method is similar to the one proposed in HSVI, but adds the notion of *selective deep sampling*.

SARSOP's selective deep sampling continues down sampling paths deeper than the HSVI collection method when doing so would likely lead to improvements in the lower bound value at belief nodes earlier in the search. The standard HSVI exploration algorithm terminates the search through belief space once the gap between the bounds is less than $\gamma^{-t}\epsilon$, where t is the current depth of the search and ϵ is an input parameter. SARSOP sampling ignores this termination when it predicts that a potential node would improve the bound, by predicting the optimal value $V^*(b)$. This prediction is done by clustering beliefs by features, such as the initial upper bound and the entropy at the belief node. The average value of belief nodes sharing the features of b is used to compute the prediction $V^*(b)$. SARSOP also implements a gap termination criterion, where sampling terminates early if the upper bound value at b, $\overline{V}(b)$ is smaller than target upper bound value for b, which is computed by the parent belief node.

3.8 Discussion

In this chapter, we have examined some of the recent point-based POMDP solvers in the literature. These methods all propose novel collection or update techniques, motivated by theoretical error bounds or simply by heuristics which intuitively should lead to useful beliefs.

When an algorithms is developed, it is generally tested against the previous algorithms in the literature. Unfortunately, these empirical comparisons tend to have a variety of issues. Most critically, when an algorithm comparison is made, it is often done between two entirely different software packages. While the computer running the packages is the same, the implementation details of the programs (optimizations in the code, language used, and many others) may differ. Secondly, the algorithms themselves often differ in multiple approaches, such as proposing a new collection scheme, a separate update method, optimized pruning techniques, improved bounds and other optimizations. Because of this, the actual reason for the planning speed improvements are obscured, since multiple parameters of the algorithms are changed all at once. Furthermore, the set of environments often differ from paper to paper, making it more difficult to come up with a full comparison, since the quality of the algorithms differ wildly when we change the domain. Finally, none of these experiments fully test all other algorithms.

Because of the shortcomings of the current empirical results, in this work we aim to provide a fuller comparison of the underlying techniques used in these algorithms. The following chapter will describe what components will be tested.

CHAPTER 4
Collection and Backup Methods for Point-Based POMDP Solvers

In this chapter, we tackle the complexity of empirically examining algorithms by breaking the algorithms up into their component parts. The goal here is to be able to compare collection methods and updating methods, while keeping all other algorithm parameters the same. This systematic approach to measuring the quality of a POMDP solver is critical for understanding which methods are applicable for what classes of domains.

As we have discussed, the key components to point-based POMDP solvers are the belief collection and belief update methods. This chapter describes in detail these methods. We show an outline linking the algorithms with their respective collection and update methods in Table 4–1. The collection and update methods are the key components of a point-based POMDP solver, as shown in the generic solver described in Algorithm 1.

Algorithm	COLLECT	UPDATE
PBVI	L_1 Norm	Full Backup
Perseus	Random	Perseus Backup
HSVI	Bound Uncertainty	Newest Points Backup
PEMA	Error Minimization	Full Backup
FSVI	MDP Heuristic	Newest Points Backup

Table 4–1: An outline of a variety of point-based solvers, where we identify the associated collection and updating methods.

We will first describe the belief collection methods that will be compared in this work. These methods are taken from the current point-based solvers in the literature. Next, we describe the methods for selecting and ordering the updates for the set of belief points. These methods are also drawn from the current research in point-based solvers.

4.1 Belief Collection

A point-based POMDP solver must specify a method for collecting belief points. Recall Algorithm 1, where we define each iteration of the belief collection step as:

$$B_{new} \leftarrow COLLECT(V_t, B, N). \tag{4.1}$$

The new set of beliefs is expanded by N points at each iteration of collection. We note that the current value function V_t and the previous set of beliefs B can all be accessed by the collection algorithm. As well, if the specific collection method requires the use of bounds, then the bounds are accessible for the collection method.

While the following algorithms do not explicitly note this, we assume that our set of belief points contains only unique points. Therefore the algorithms check that any belief does not already exist in the set B before adding it.

4.1.1 Random Collection

Using random exploration to collect belief points is probably the simplest method to collect a large set of reachable belief points and was first proposed in Perseus [52]. Here we present a random belief collection method (Algorithm 2) which collects new points at each new collection iteration in the point-based algorithm. This is slightly different from the collection method proposed in Perseus, which collects one initial batch of beliefs at the start of the algorithm. In this work, we collect new belief points at each collection step, to keep this method similar to all other collection methods.

Algorithm 2 Random Collection Algorithm(V, B, N)

```
$B_{new} \leftarrow \emptyset$
**while** $|B_{new}| < N$ **do**
    $b \leftarrow b_0$
    $s \sim b$
    random-explore($s$, $b$)
**end while**
**return** $B_{new}$
```

Algorithm 3 random-explore(s, b)

```
**if** current exploration is not terminated **and** $|B_{new}| < N$ **then**
    $a \leftarrow random(A)$
    $s' \sim T(s, a, s')$
    $z \sim O(s, a, z)$
    $b' \leftarrow \tau(b, a, z)$
    random-explore($s'$, $b'$)
    $B_{new} \leftarrow B_{new} \cup b'$
**end if**
```

This collection algorithm produces random traces through belief space by selecting uniformly random actions at each step, while maintaining the underlying state. It iterates through traces until the required number of belief points have been found. We also start a new trace when the current exploration is terminated. The criterion for termination could be, for example, if the state has not changed for some fixed number of steps, or simply some large fixed number of steps.

An advantage of random collection is that it is quite efficient, taking only $O(|S|^2 + |A| + |Z|)$ time at each step, $O\Big(|N|(|S|^2 + |A| + |Z|)\Big)$ for a full collection.

Note that the belief point is added after the $random - explore$ recursion call, which means the belief points will be added in reverse order (i.e. the last belief point in the exploration will be first. This is done deliberately, and will be discussed in the belief point update section.

4.1.2 MDP Heuristic Collection

We use the collection method from FSVI as our MDP Heuristic collection method. The intuition for this collection method is to use the optimal MDP action selection as a heuristic to guide the search through belief space.

Algorithm 4 MDP Heuristic Collection Algorithm(V, $\overline{V}$, B, N)

```
$B_{new} \leftarrow \emptyset$
**while** $|B_{new}| < N$ **do**
    $b \leftarrow b_0$
    $s \sim b$
    mdp-explore($s$, $b$)
**end while**
**return** $B_{new}$
```

As in the other exploration-style collection algorithms, we iterate over belief traces until we collect N belief points.

This collection method requires solving the underlying MDP, which is $O(|S|^2|A||N|)$ for N iterations of value iterations. However, this process only needs to be done once, and is computationally fast compared to the rest of the algorithm.

Algorithm 5 mdp-explore(s,b)

```
if current exploration is not terminated and $|B_{new}| < N$ then
    $a \leftarrow \operatorname{argmax}_a Q(s,a)$
    $s' \sim T(s,a,s')$
    $z \sim O(s,a,z)$
    $b' \leftarrow \tau(b,a,z)$
    mdp-explore($s'$, $b'$)
    $B_{new} \leftarrow B_{new} \cup b'$
end if
```

An advantage to using a simple heuristic to guide the search is that the resulting collection algorithm is computationally efficient. The algorithm takes $O(|S|^2 + |A| + |Z|)$ time at each step, the same as the random exploration heuristic. A full collection therefore takes $O\Big(|N|(|S|^2 + |A| + |Z|)\Big)$ operations.

4.1.3 L_1 Norm Collection

The L_1 norm collection algorithm is based on the PBVI collection implementation. The goal in this collection method is to discover the most distant belief points at each step, in a L_1 distance sense. The algorithm is described in Algorithm 6.

We define $||b' - B||_1$ (the norm between a belief point b' and a set of beliefs B) as the distance $||b' - b_{closest}||_1$ where $b_{closest} \in B$ is the closest (L_1 norm) belief point in B to b'. The result represents the distance from the belief b' to the entire set B.

We also define $O_{belief}(b,a,z)$ as the probability of observing z when executing action a while in belief b:

$$O_{belief}(b,a,z) = \sum_{s \in S} b(s) O(s,a,z). \tag{4.2}$$

Algorithm 6 L_1 Norm Collection Algorithm(V, B, N)

```
$B_{new} \leftarrow \emptyset$
while $|B_{new}| < N$ do
    $b \leftarrow random(B)$
    $max \leftarrow 0$
    for $a \in A$ do
        $z \sim O_{belief}(b, a, z)$
        $b' \leftarrow \tau(b, a, z)$
        if $||b' - B||_1 > max$ then
            $b_{max} \leftarrow b'$
            $max \leftarrow ||b' - B||_1$
        end if
    end for
    $B_{new} \leftarrow B_{new} \cup b_{max}$
end while
return $B_{new}$
```

This collection method is the collection method used in PBVI, although we modify the method to allow a fixed number of new beliefs. While PBVI creates a successor belief for each $b \in B$, doubling the size of the set at each step, here we only increase the set by N, by randomly picking N parent beliefs. This lets us limit the expansion of the belief set, to avoid problems in domains with a long required planning horizon.

The running time of the collection algorithm presented here is $O\Big(|A|(|S||B| + |Z|)\Big)$ for a single collected belief point, giving the running time for an entire iteration as $O\Big(|N||A|(|S||B| + |Z|)\Big)$.

One disadvantage of the L_1 norm approach is that it does not use the rewards of the environment to guide the belief expansion. Since belief points are picked uniformly as parent beliefs, it is likely that the belief point successors will be concentrated in densely explored areas of the belief space. This can be problematic in domains where a long planning horizon is required. We attempt to remedy this issue in a modification of this collection algorithm L_1 Leaf Biased Collection.

4.1.4 L_1 Leaf Biased Collection

In this work, we present a new algorithm, the L_1 Leaf Biased collection method, an extension to the L_1 Norm collection method. This algorithm was created as a response to limitations of the L_1 Norm collection method in domains which require long planning horizons.

In the L_1 Norm collection, N points are randomly sampled from the set to be the parent belief of a new belief point. Since we draw the points uniformly, the chosen beliefs are likely to be in areas of the belief space that are the most dense. If this is the case, then the new belief point will also be near this dense space, which compounds the problem. If we have a domain where the shortest distance between the start state and the closest reward is some relativity large number of steps, then it can be very difficult for L_1 Norm collection to collect a belief near the reward. While the area near the start state might be quite dense with sampled beliefs, areas distant from the start may have very few. We note that this problem does not exist as strongly in the original PBVI collection method (which collects a successor belief for each current belief). However, in our long-horizon domain, the PBVI collection will have $B = O(2^t)$ when it has reached a planning horizon of t.

To remedy this problem, we developed L_1 Leaf Biased collection, which biases the random selection of belief points towards beliefs which have not yet had a successor. With chance l, the parent belief is sampled only from B_l, where $B_l = \{b \mid b \text{ has no successor }, \forall b \in B\}$. Otherwise, with chance $(1 - l)$, b is sampled from B as usual. This modification is in the spirit of the L_1 collection method, and should help guide the search towards areas which are less densely explored.

We also add a second modification to L_1 Leaf Biased collection which expands the set of possible new belief points. For each action, instead of sampling the observation from the observation model, we iterate though all observations and return the best belief overall. The advantage is that this gives us a wider set of possible new beliefs, which could give the algorithm a farther (L_1 norm) new belief points. The tradeoff is that we must compute the distance for all observations. The full algorithm is detailed in Algorithm 7.

The running time for this algorithm is $O(|S||A||Z||B|)$ for a single collected belief point, giving the running time for an entire iteration as $O(|N||S||A||Z||B|)$.

4.1.5 Bound Uncertainty Collection

The Bound Uncertainty Collection algorithm chooses beliefs where the difference between the upper and lower bounds of the value function are highest, thereby adding beliefs which will reduce the error between the bounds the most. This is the collection algorithm used in HSVI. We outline the method in Algorithm 8. Note that since this algorithm requires an upper bound $\overline{V}$, it takes in $\overline{V}$ as a parameter.

Algorithm 7 L_1 Norm Leaf Biased Collection Algorithm(V, B, N)

```
$B_{new} \leftarrow \emptyset$
while $|B_{new}| < N$ do
    $r \leftarrow rand(0, 1)$
    if $r < l$ then
        $b \leftarrow random(B_l)$
    else
        $b \leftarrow random(B)$
    end if
    $max \leftarrow 0$
    for $a \in A$ do
        for $z \in Z$ do
            $b' \leftarrow \tau(b, a, z)$
            if $||b' - B||_1 > max$ then
                $b_{max} \leftarrow b'$
                $max \leftarrow ||b' - B||_1$
            end if
        end for
    end for
    $B_{new} \leftarrow B_{new} \cup b_{max}$
end while
return $B_{new}$
```

Algorithm 8 Bound Uncertainty Collection Algorithm(V, B, N)

```
$B_{new} \leftarrow \emptyset$
while $|B_{new}| < N$ do
    bound-explore($b_0$, 1)
end while
return $B_{new}$
```

Algorithm 9 bound-explore(b,t)

```
if $|B_{new}| < N$ and $excess(b,t) > 0$ then
    $a \leftarrow \operatorname{argmax}_{a \in A} Q^{\overline{V}}(b,a)$
    $z \leftarrow \operatorname{argmax}_{z \in Z} \Big[P(z|b,a)excess\Big(\tau(b,a,z), t+1\Big)\Big]$
    $b' \leftarrow \tau(b,a,z)$
    explore($b'$,$t+1$)
    $B_{new} \leftarrow B_{new} \cup b$
    $\Upsilon_{\overline{V}} \leftarrow \Upsilon_{\overline{V}} \cup (b, \max_{a \in A} Q^{\overline{V}}(b,a))$
    $t \leftarrow t+1$
end if
```

The function $excess(b,t)$ is defined in Equation 3.12, and see Equation 3.11 for a definition of $Q^{\overline{V}}(b,a)$. The upper bound is represented as a point-set $\Upsilon_{\overline{V}}$ which maintains the belief points and their associated upper bound values. Computing the upper bound $\overline{V}(s)$ is done the same as in HSVI, via an approximate projection onto the convex hull of our point-set $\Upsilon_{\overline{V}}$. Points are added to $\Upsilon_{\overline{V}}$ in reverse order, much as in the belief point collection.

This algorithm iterates over explorations through the belief space, with transitions following the specified action and observation heuristics. We terminate the collection when N belief points have been collected.

The $excess(b,t)$ formula requires the computation of the upper and lower bound at the point b. The resulting running time of $excess(b,t)$ is $O\Big(|S|(|\Upsilon_{\overline{V}}| + |\Gamma_V|)\Big)$, which gives a running time of $O\Big(|A|+|Z||S|(|S|+|\Upsilon_{\overline{V}}|+|\Gamma_V|)\Big)$ for a single new belief points. Therefore, the full collection step takes $O\Big(|N|\big(|A|+|Z||S|(|S|+|\Upsilon_{\overline{V}}|+|\Gamma_V|)\big)\Big)$ time.

4.1.6 Error Minimization Collection

The Error Minimization Collection is adapted from the Point Based Error Minimization (PEMA) algorithm. We describe the collection algorithm in Algorithm 10. We have modified the PEMA collection algorithm by having it collect N belief points per iteration instead of one new belief per step. Each new collected belief is considered for being the parent belief point for the next belief.

Algorithm 10 Error Minimization Collection Algorithm(V, $\overline{V}$, B, N)

```
$B_{new} \leftarrow \emptyset$
while $|B_{new}| < N$ do
    $max \leftarrow -\infty$
    for $b \in B \cup B_{new}$ do
        if $b$ has unexplored children then
            if $\overline{\epsilon}(b) > max$ then
                $max \leftarrow \overline{\epsilon}(b)$
                $b_{max} \leftarrow b$
            end if
        end if
    end for
    $b'$ is calculated as in Equation 3.18
    $B_{new} \leftarrow B_{new} \cup b'$
end while
return $B_{new}$
```

This collection algorithm is quite computationally intense, since the potential error heuristic $\overline{\epsilon}(b)$ must be calculated for most beliefs. The running time for $\overline{\epsilon}(b)$ is $O(|S|^3|A||Z||B|)$, so a single new belief point takes $O(|S|^3|A||Z||B|^2)$ time. A full iteration, collecting N new points will take $O(|S|^3|A||Z||B|^2|N|)$ time.

4.2 Belief Updates

Once the beliefs have been collected, the point-based solvers need to specify which set of belief points will be updated, and in what order. This is shown in the generic point-based structure, Algorithm 1 (Section 3.1), where we allow U iterations of backups:

$$V_{t+1} \leftarrow UPDATE(V_t, B, B_{new}). \tag{4.3}$$

Note that we allow the update algorithm to use two separate sets of beliefs, the entire collected belief set and the belief set that was just collected.

In the exploration-based collection algorithms (Random, MDP Heuristic, Bound Uncertainty) we always add the belief point to the set after the recursive call, which means the belief points are added starting from the end of the trace, with the belief point at the start of the exploration added last. We note that the *backup*(b) operation uses the value estimates of the successor beliefs b' to compute the current value estimate (Equation 3.4). Thus we will get better estimates for a given b if the values for the successor beliefs b' are already updated. Therefore we backup the belief points in the trace in reverse order to maximize the speed of the value updates.

4.2.1 Full Backup

In a Full Backup, we execute *backup*(b) on each belief in the full set of belief points, including the ones just collected, i.e. $B \cup B_{new}$. The advantage of this approach is that it will not waste collected belief points by only updating them once.

Backing up a belief point multiple times is advantageous since the values of the successor beliefs will likely be updated in between. This will give a resulting higher $V(b)$ at the original belief, even if it has already computed α-vectors. The disadvantage is that it is computation expensive, since the *backup* operation is $O(|S|^2|A||Z||\Gamma|)$. This is the backup method used in PBVI and PEMA.

4.2.2 Newest Points Backup

In Newest Points Backup, we execute *backup*(b) only on $b \in B_{new}$. By only backing up the newest points, we save a lot of time in the backup phase, since the number of new belief points is fixed at each iteration. As well, the points backed up are guaranteed to not have constructed α-vectors before, making it more likely for the point to support a useful α-vector.

However, this approach ignores all the belief points that have been collected previously, which could support new useful α-vectors. The argument for this method is that the extra time not spent backing up older beliefs could be better used to collect and update new belief points. This method is the version used in $HSVI$ and $FSVI$.

4.2.3 Perseus-style Backup

The Perseus-style backup operation guarantees that the value for all belief points is improved or at least maintained, while only backing up a subset of the full belief set B. This method was introduced in the Perseus algorithm.

This method randomly picks a point b from our entire belief set and then adds the associated α-vector *backup*(b) to our set if the new α-vector improves the value

Algorithm 11 Perseus-style Backup Algorithm(V_t, B, B_{new})

```
$k \leftarrow t$
$B_{full} \leftarrow B \cup B_{new}$
$\tilde{B} \leftarrow B_{full}$
while $|\tilde{B}| > 0$ do
    $b \leftarrow random(\tilde{B})$
    $\alpha \leftarrow backup(b)$
    if $\sum_{s \in S} b(s)\alpha(s) > V_t(b)$ then
        $\Gamma_{k+1} \leftarrow \Gamma_k \cup \alpha$
    end if
    $\tilde{B} \leftarrow \tilde{B} - \{b\}$
    for $b' \in \tilde{B}$ do
        if $\sum_{s \in S} b'(s)\alpha(s) > V_t(b)$ then
            $\tilde{B} \leftarrow \tilde{B} - \{b'\}$
        end if
    end for
    $k \leftarrow k + 1$
end while
return $V_k$
```

at b. It then checks if this new α-vector has improved the value of any other belief b', and if it has, removes the belief b' from the beliefs to be considered for updates. Therefore, this method attempts to improve every belief point b in the full set, while requiring fewer backups than the full backup method.

4.2.4 Pruning

As we have discussed in Section 2.2.4, one major limitation of exact POMDP solvers is the large number of α-vectors that must be maintained. Here we describe optimizations used to reduce the number of α-vectors that must be maintained.

For each belief point b, we compute the α-vector $\alpha = \mathit{backup}(b)$. However, instead of directly adding α to Γ, we check that the vector has improved or maintained the value at the current point b, i.e.:

$$\left[\sum_{s \in S} b(s)\alpha(s)\right] \geq \left[\sum_{s \in S} b(s)V(s)\right]. \tag{4.4}$$

We use this method to immediately prune α-vectors that are most likely dominated by some other α-vector. It is possible that we will prune α-vectors which would be optimal for some $b' \neq b$. However, this optimization provides a large speedup by immediately pruning many dominated vectors.

Once the α-vector has passed this check, we check if it dominates any other single α-vector already in Γ. If a α-vector α' is dominated by the new α-vector, then it is removed from the set Γ. This pruning will not remove an α-vector which is dominated by a set of other α-vectors, because checking whether an α-vector is dominated by multiple α-vectors is expensive whereas doing a single-vector check is fast ($O(|S|)$ for a single domination check).

The algorithms which update the value at every belief point (PBVI, Perseus, PEMA) do not require α-vector pruning, since at every iteration they start with a fresh value function ($\Gamma_t = \emptyset$). This is possible since the value at each belief point $b \in B$ is guaranteed to be at least maintained. However, there are issues with that approach, for the following reasons. First, α-vectors from the previous iteration may give a higher value for some $b \notin B$, but will not be present in the newest iteration of Γ. Since the α-vectors from the previous set are not carried over, these values will be lost. Second, this setup performs the *backup* operation using only the α-vectors from the previous iteration, so as the algorithm iterates through B, the backups only use the previous set of α-vectors Γ_{t-1}, and not the α-vectors collected in the current iteration. As many belief point updates are usually done in each iteration, using the recently (current iteration) created α-vectors in the *backup* operation leads to substantial speedups.

Because of these reasons, in the empirical analysis in this work we build one set of α-vectors (Γ), and each backup operation is able to draw from all $\alpha \in \Gamma$. We keep the overall size of Γ low via the domination-based pruning technique described above.

4.3 Summary

In this chapter, we have separated the key components of point-based POMDP solvers, the belief collection process and the belief updating process, from the original algorithms in the literature. Our focus now is to compare these methods, as opposed to comparing the original algorithms, which may vary in multiple respects. This approach is crucial for understanding the advantages and disadvantages of these component methods.

CHAPTER 5
Experimental Results

The primary goal of this thesis is to gain a better understanding of the strengths and weaknesses of the methods used in point-based POMDP solvers. In this chapter, we present experimental results which will demonstrate the viability of different approaches in a variety of domains. We test the belief collection and belief update methods we have previously discussed, as well as examine the influence of other parameters.

5.1 Domains

We conduct our experiments on a variety of domains, mainly drawn from the standard benchmark problems in the POMDP literature. Here we describe the domains and their parameters. We show the size of the domains in Table 5–1.

Hallway

Hallway [28] is a classic small robot navigation domain, where the robot starts in a random location in the environment, and must navigate to a goal state. It should be noted that we modified the POMDP specification file to have the agent transition to an end state after reaching the goal, instead of transitioning to a uniform distribution (and thereby restarting the episode). If a terminal state is hardcoded

Domain	#States	#Actions	#Observations
Hallway	61	5	21
Tag	870	5	30
RockSample[5,5]	801	10	2
RockSample[7,8]	12545	13	2
FieldVisionRockSample[5,5]	801	5	32
Underwater Navigation	2653	6	102
Dialogue	29	31	28

Table 5–1: A chart of the domains used and the size of their state, action and observation spaces, respectively.

with the original definition, the value function (based on the model) will be higher than the actual achievable averaged discounted reward (ADR), which can lead to suboptimal policies. This is because the model itself has no knowledge of a hardcoded terminal state, so the planner is solving a slightly different task.

There are several other benchmark problems from the literature [28] that were proposed around the same time as Hallway. However, many of these domains have ceased to be useful as benchmarks due to their small size and the increasing efficiency of the point-based solvers.We include the Hallway domain as a representative of this class of early POMDP domains.

Tag

Tag [39] is a robot navigation domain where the agent must maneuver in an environment while attempting to catch an opponent which follows a stochastic fixed policy. The agent receives a reward if it is able to "Catch" the opponent when it is in the same location. While the actual state space is not small ($|S| = 870$), part of the state space is the robot's position which is fully observable. The belief state is effectively only over the position of the opponent, which is much smaller (< 30 states). Tag has been widely used as a test-bed for point-based POMDP solvers.

RockSample

RockSample is a scalable navigation problem that models rover exploration [48]. The agent receives rewards by travelling to rocks in the environment, and sampling them if they are "good", which is detected by a "Check" action. The probability of an observation being correct increases as the agent gets closer to the rock. An instance of RockSample with a map size $n \times n$ and k rocks is denoted as RockSample$[n, k]$. We have modified the original specification by not giving a reward when moving into the exit area for convenience. RockSample is an interesting domain to include since it is not goal oriented. In this kind of domain, there is room for the agent to build an iteratively better policy.

FieldVisionRockSample

We use the FieldVisionRockSample (FVRS) domain from Ross and Chaib-draa [42]. This domain is a modification of the RockSample domain, where the noisy sensor is able to detect all rocks at each step. This increases the observation space but reduces the number of available actions in comparison to the original RockSample. The decrease in the number of actions is caused by the removal of the "Check" actions.

Underwater Navigation

The Underwater Navigation domain [26] is an instance of a coastal navigation problem. The agent must traverse the environment while avoiding dangerous rocks. However, the robot has a poor idea of its own location. It must move to the sides of the environment to localize itself. This domain has a large state and observation space. However the domain is simple in some ways, in that the transitions and observations are deterministic. A primary difficulty in the Underwater Navigation domain is that there is substantial aliasing since the state space is much larger than the observation space. As well, the required planning horizon is rather large.

Dialogue

The Dialogue domain is a POMDP model used for dialogue management between a user and an intelligent wheelchair. It is a modification of the POMDP model described in [4]. In this domain, the user has an (unknown) intent for what action the robot should execute. The agent can execute a "command" action, receiving a positive reward if correct and zero reward otherwise. The agent can also execute one of three "query" actions, which are strictly information gathering. These queries return observations related to the user's intent. We include this domain as an example of a task with predominantly information gathering actions. The POMDP model is available online [1].

5.2 Experimental Method

Our goal in the following experiments with the POMDP solvers is to measure the quality of the policy returned by a given POMDP planner. This is done by measuring averaged discounted return (ADR), which we defined in Equation 2.2. This can be estimated by simulating trials within the environment, where the agent starts in a state drawn from the start state distribution and follows the policy returned by the POMDP planner for some fixed number of steps. We compute the discounted return from this trial, and average over a number of simulated trials. We are also interested in seeing how much computation time the POMDP planner requires to produce a policy that is optimal for the domain, and, otherwise, at what ADR does the POMDP planner converge.

In our results, we present the quality of the resulting policies given a range of different planning times. This approach not only lets us see when a given algorithm has converged, but will also let us evaluate the speed of the algorithms, to see which ones are able to quickly approach a good solution. In this setup, the quality of a policy for a given available planning time t is the best policy produced before t. We make this stipulation since we only compute policies after a full iteration has completed, so the policy for a given time t is the policy computed by the value function created immediately before time t.

All experiments were run on a 2.2Ghz Opteron with 8GB of system memory. The averaged discounted return (ADR) is computed by simulating the policy on the environment for 500 trials. The following results were computed by averaging over several POMDP solutions, where the number of solutions depended on the domain.

This is done because several of the solvers have stochastic components (for example, in the belief sampling). The Hallway domain used 30 trials, all RockSample domains and the Tag domain used 20 trials, and the Underwater Navigation and Dialogue domains used 10 trials.

The ϵ parameter for Bound-Uncertainty collection is set to $\epsilon = .001$. The l parameter for L_1 Leaf Biased collection is set to $l = .75$. Empirically, the results are not very sensitive to this parameter, and .75 was chosen for performance. For clarity, we omit the confidence intervals in most of our results, however, they are displayed in the results in Section 5.6.

5.3 Belief Collection and Belief Update Ordering

In our first set of experiments, we compare the belief collection methods while also varying the belief update ordering method. Our primary goal is to see if we can draw any overall conclusions on how the different collection and updating methods affect the speed of convergence, as well as seeing whether this is domain-dependent.

Since in this phase of the experiments we are comparing the collection and update methods simultaneously, we try to keep the rest of the parameters as similar as possible. However, since the Error Minimization collection algorithm (from PEMA) is, relative to the other methods, much slower to collect beliefs, it does poorly with the large number of belief points usable in the other collection methods. Therefore the number of belief points (N) collected at each step in Error Minimization is less than in all other collection methods. Unless otherwise stated, all belief collection method use $N = 100$, except for Error Minimization which always uses $N = 10$. In all cases, we do a single round of belief point updates per iteration ($U = 1$).

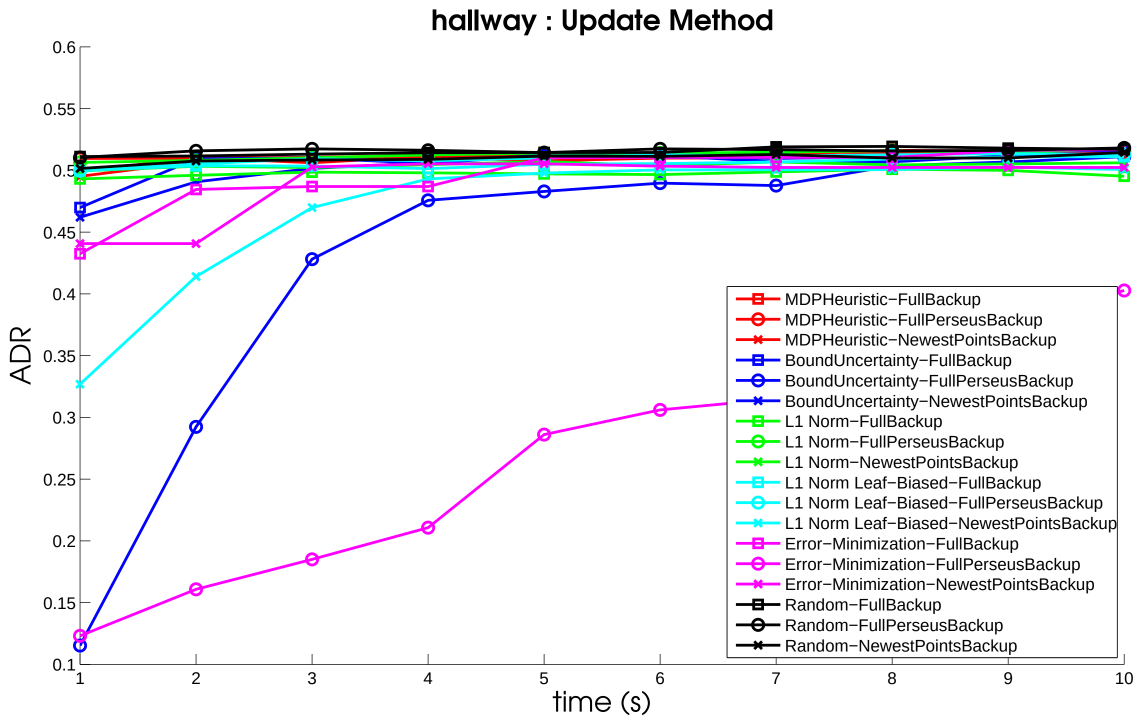

Figure 5–1: The Hallway domain. In this experiment, we compare different belief collection methods and different belief update methods.

We present the Hallway domain in Figure 5–1. At each interval, 10 belief points were collected ($N = 10$).

The first thing to note is how quickly these solvers find the optimal solution in this small domain, with most finding the optimal solution in less than one second. We therefore use Hallway as an example of early benchmarks, but we concentrate our experiments on more recent domains.

The Error Minimization belief selection, coupled with Perseus-style backups is much slower to converge than all other methods. This is caused by a combination of factors. First, the Error Minimization collection method is slower than the other belief collection methods ($O(|S|^3|A||Z||B|^2)$ per belief point). Secondly, the Perseus-style backup method attempts to speed up the planning by cutting the number of updates done per step. However, the resulting policy at each step will in general not be as good as in a full backup over all belief points. While this is an acceptable tradeoff if the algorithm is mainly time-constrained by the belief updates, this setup can do poorly when the collection method is computationally intensive. In the Error Minimization-FullPerseusBackup case, too much time is spent doing belief collections compared to belief updates, which causes the algorithm to take a longer time to come to a solution.

We present the results for the Tag domain in Figure 5–2. We see that the choice of collection method is not critical in the Tag domain. Even the simplest collection method, Random exploration, is able to find an optimal solution as fast as any other collection method. We can attribute that to the domain specifics. In the Tag domain, the physical environment containing the robot is not large, so the required horizon to find the opponent is relatively short. Also, the area where the agent receives a reward (when it is able to catch the opponent) is relatively large in belief space, so it is not difficult for even simple collection methods to find this area of the belief space. Error Minimization belief selection coupled with the Newest Points update method is slowest to converge. The Newest Points method only updates a small subset of

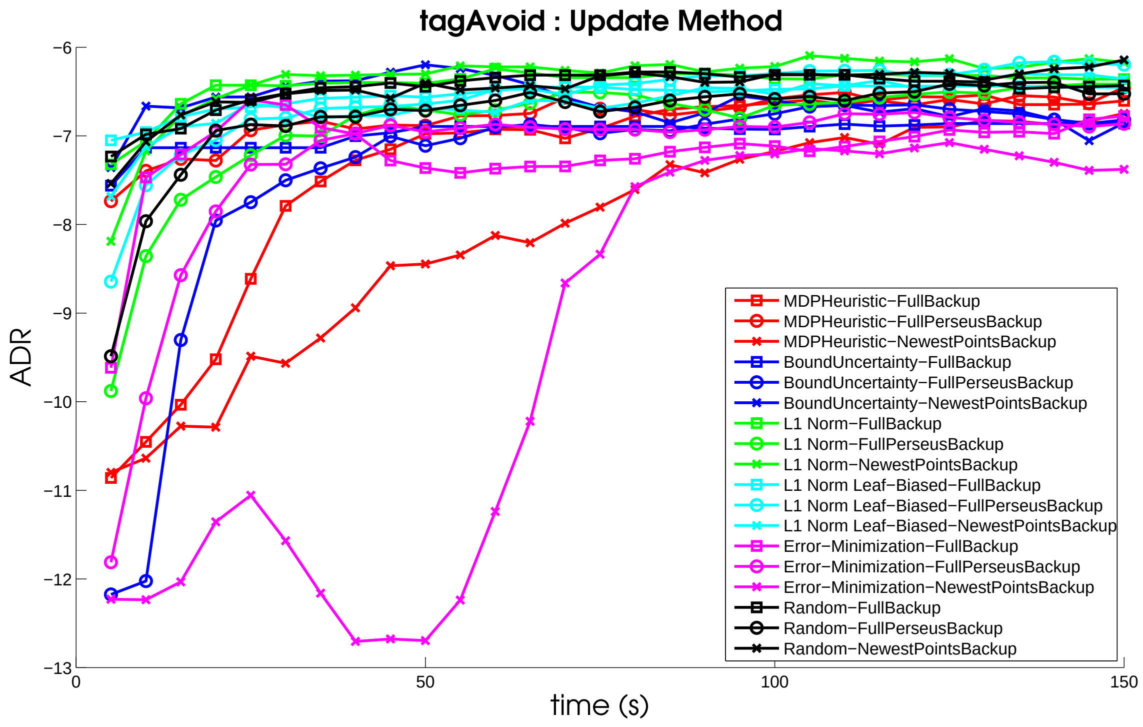

Figure 5–2: The Tag domain. In this experiment, we compare different belief collection methods and different belief update methods.

the current belief points, so, much like the Perseus backup method, it does better when the collection is quite fast.

With the exception of the Error Minimization collection method, there is little difference as we vary the update method. In addition, the advantage of one update method over another is not consistent across the collection methods, so little can be concluded.

Figure 5–3 shows our results for the RockSample[5,5] domain. In general, for this domain, the choice of update method-whether a Full Backup, the Newest Points Backup, or Perseus-style Backup makes little difference. However we see a small

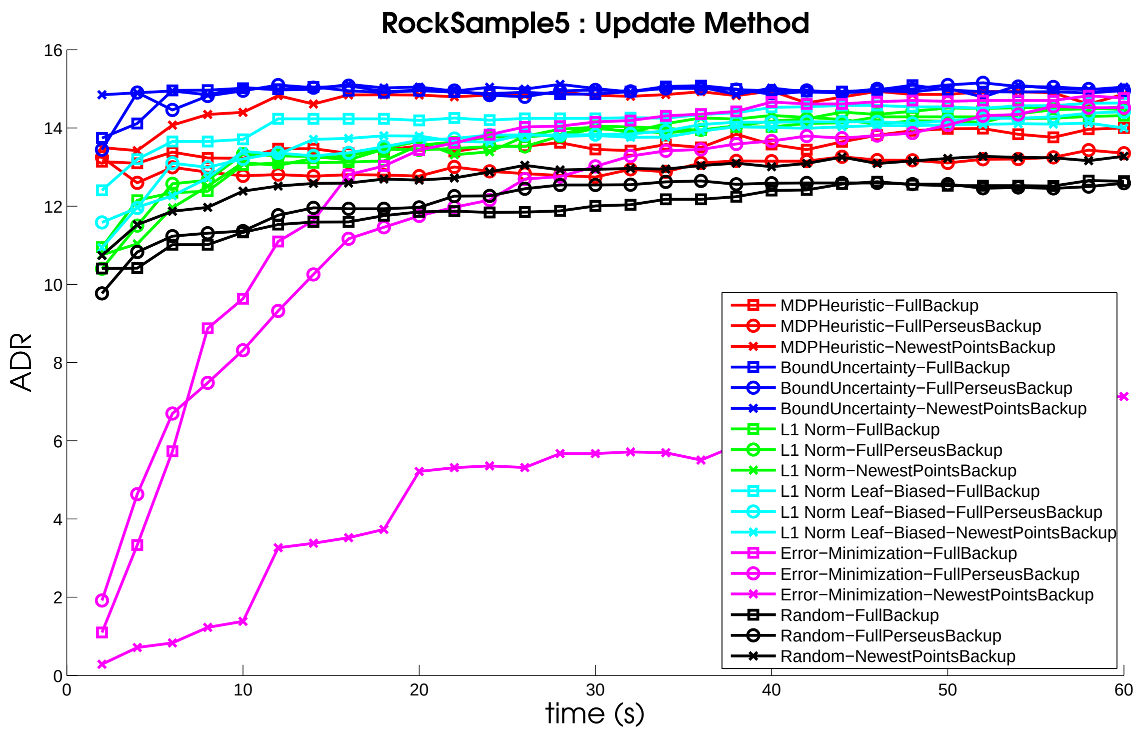

Figure 5–3: The RockSample[5,5] domain. In this experiment, we compare different belief collection methods and different belief update methods.

separation in the speed of convergence when we vary the collection method. The MDP Heuristic and Bound Uncertainty collection methods seem to be slightly more efficient. Both of these methods have an optimistic bias for their exploration. In the RockSample domain, the target rocks are fully observable, and the quality of the observation improves as the robot approaches the rock. Therefore it is understandable that these optimistic approaches do well. The Random Collection does not do as well in this domain, presumably because it is less likely to find the useful belief in this domain through random action selection. The domain has a relatively higher number of actions (compared to most domains examined in this thesis), and to do well, the robot needs to select the correct "Check" action when it is near the associated rock. Selecting the correct action may be difficult to find through a random exploration. The L_1 Norm Leaf-Biased modification shows a slight improvement over the standard L_1 Norm exploration. This should be attributed to the modification in L_1 Leaf-Biased which biases the collection towards more distant beliefs, which should help it find the small areas in belief space which are useful for a good policy. Finally, we see that the Error Minimization collection method, while slower than the other methods, is able to get a near-optimal policy faster than many of the other collections methods, when not using the Newest Points Backup.

We also notice that the methods have not converged to the same value after 60s of planning time. This is largely due to the time period tested, since in the long term, most collection method are guaranteed to converge to optimal, as discussed in Section 5.10. We see that most methods are able to find quite a good solution very quickly, but some take a longer time to refine it to the optimal.

As would be expected, the results for RockSample[7,8] (shown in Figure 5–4) are similar to the results from RockSample[5,5]. The fact that the domain is substantially larger causes the difference in effectiveness in the collection methods to be exaggerated. Even after substantial planning time, the final policy produced by the optimistic heuristics (MDP Heuristic and especially Bound Uncertainty) are better than the other collection methods. In this domain, the Leaf Biased modifications do not give an advantage to the L_1 Norm collection method, which we discuss further in Section 5.7.

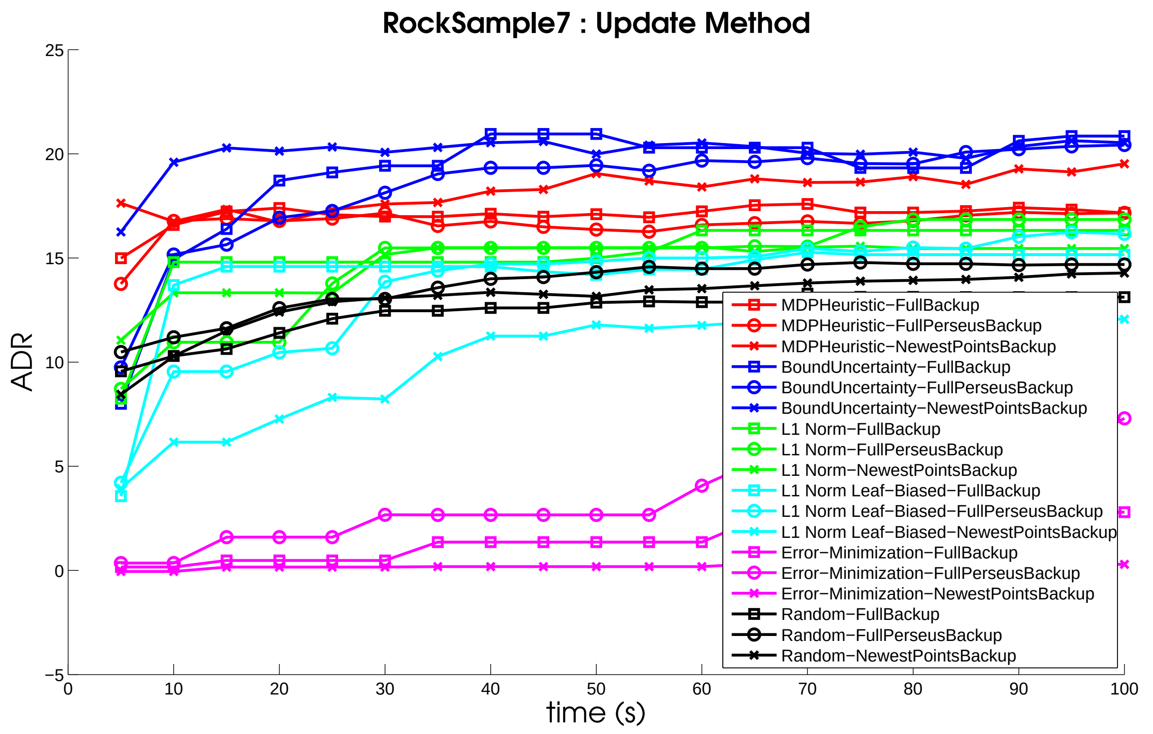

Figure 5–4: The RockSample[7,8] domain. In this experiment, we compare different belief collection methods and different belief update methods.

Figure 5–5 shows the results for the FieldVisionRockSample[5,5] domain. The domain has a much larger observation space than standard RockSample, but this does not seem to negatively impact the efficiency of the algorithms. Again, the algorithms tend to be able to find a reasonable policy quite quickly, and very slowly are able to refine it. The order of convergence is essentially the same as for standard RockSample.

We show the results from the Underwater Navigation domain in Figure 5–6. While still a robot navigation task, this differs from the previous domains in several important ways. It is primarily “goal-based” in that once the agent has found a good

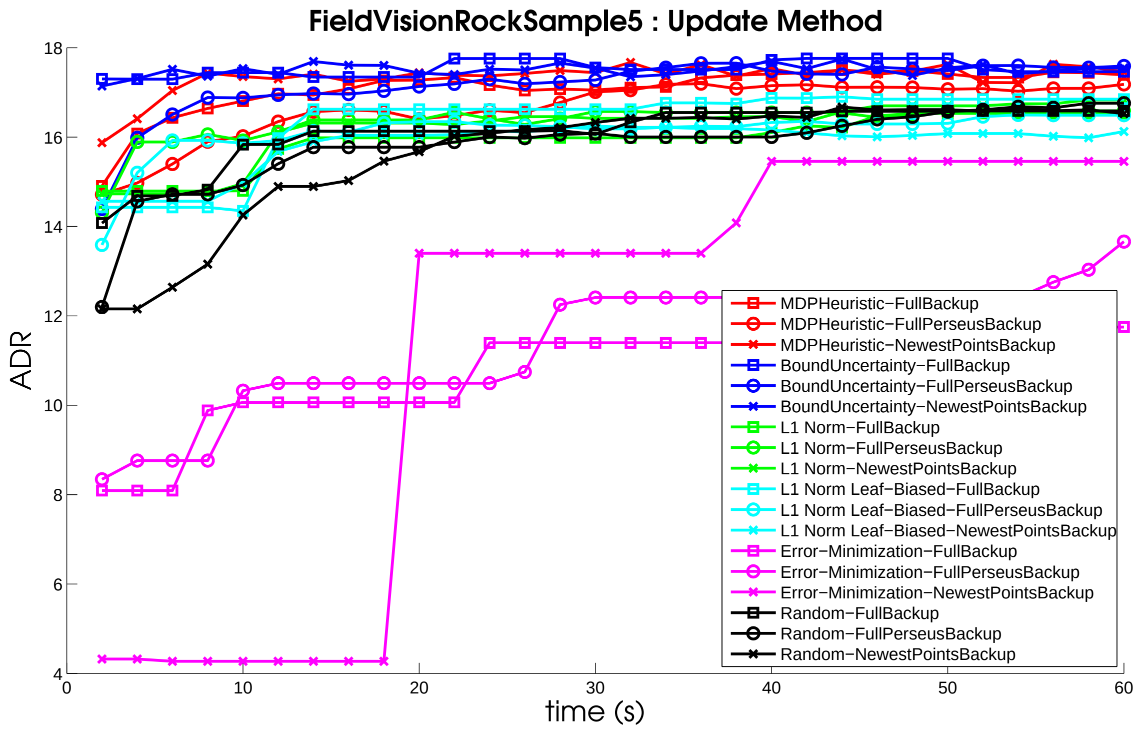

Figure 5–5: The FieldVisionRockSample[5,5] domain. In this experiment, we compare different belief collection methods and different belief update methods.

policy, there is very little to refine. Additionally, it has a long planning horizon, taking many (> 40) steps to find the first rewards.

The optimistic collection methods (Bound Uncertainty and MDP Heuristic) are generally able to converge extremely quickly. This can be attributed to the fact that using the MDP solution makes it very easy to find the (distant) goal states. We also note that the Random Collection is still able to do relatively well. We see that the L_1 Norm Leaf Biased is starting to converge, whereas standard L_1 Norm is never able to find a solution. We attribute this to the substantial planning horizon required in Underwater Navigation. We explore this more fully in Section 5.7.

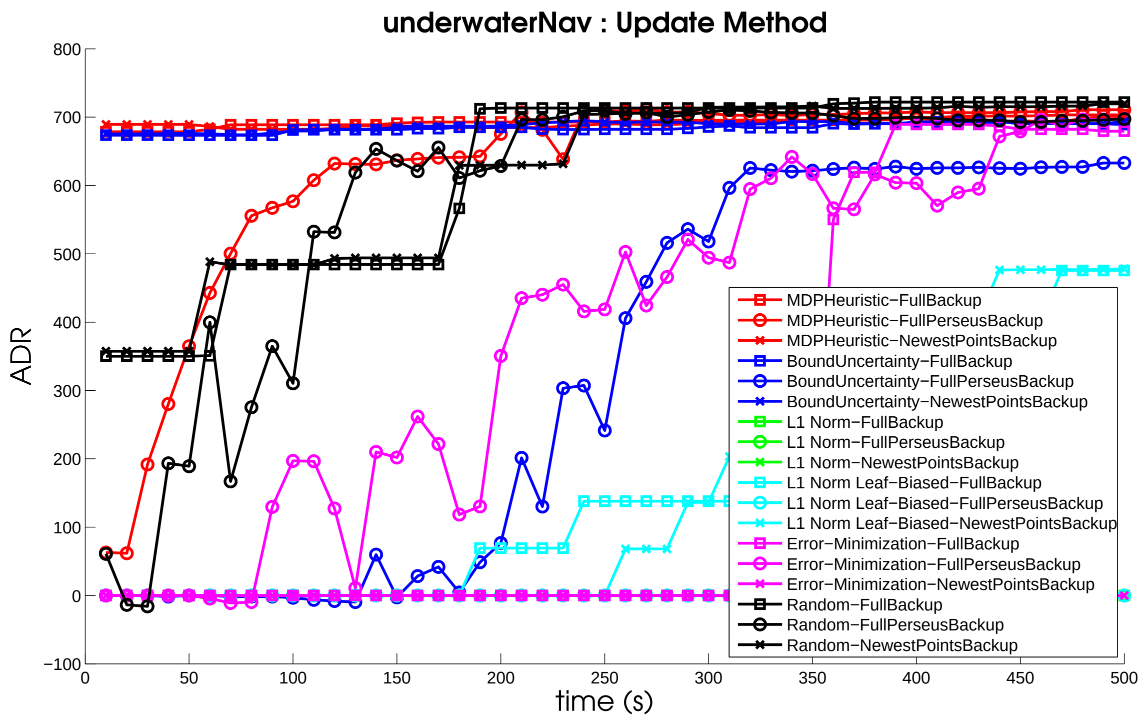

Figure 5–6: The Underwater Navigation domain. In this experiment, we compare different belief collection methods and different belief update methods.

In this domain, the choice of update method used has a large effect on the resulting policy. When using the optimistic collection methods, the algorithms converge much faster with a full backup or a newest points backup, in comparison to when using a Perseus backup. These collections methods are able to find the critical belief points quickly. When this occurs, it seems it is better to guarantee that they are updated (as in the Full Backup and Newest Points Backup). In the Perseus case, it is not guaranteed that these points will be updated, only that the value at these points are at least maintained. It is possible that these important beliefs have their values improved by new α-vectors, but the values are not improved nearly as much as they could if an α-vector were to be created at that belief.

This advantage does not seem to generalize over all collection methods, however. In the Random collection case, it does not seem to matter which update method is used. When using the Error Minimization collection, it seems that Perseus starts converging slightly quicker. Since it takes a long time for Error Minimization to reach the critical beliefs, we might attribute this to the savings on the time to update the beliefs. However, when Error Minimization is coupled with Full Backup, it converges to the optimum around the same time as when using the Perseus Backup.

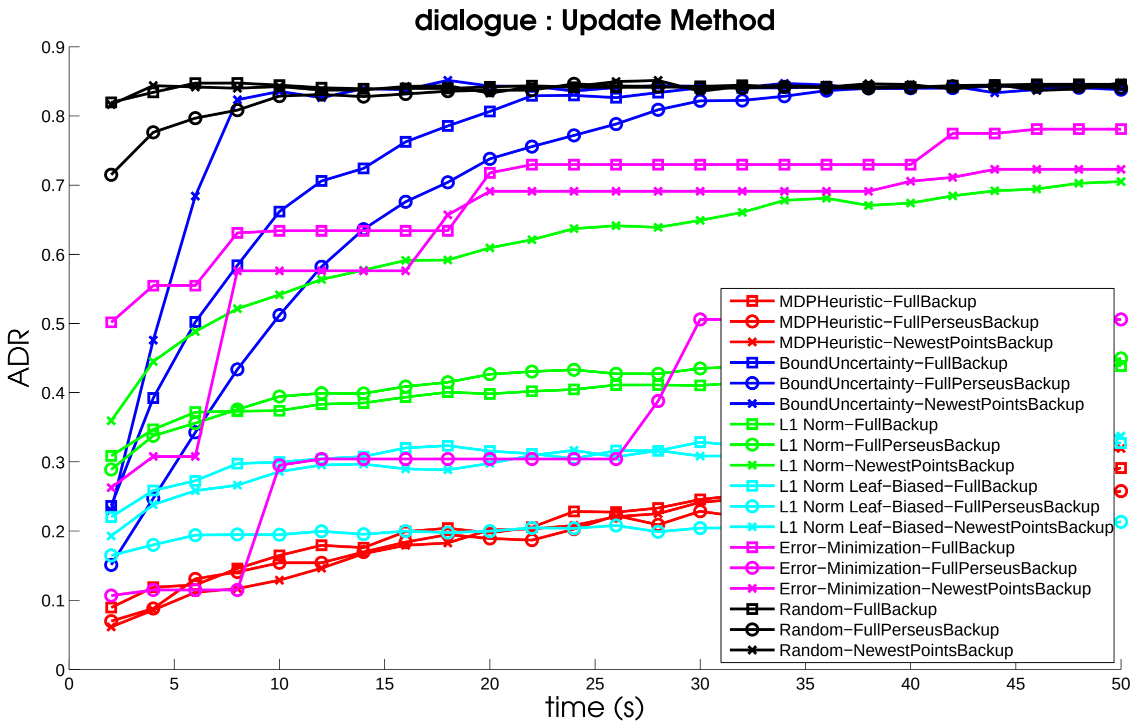

Figure 5–7: The Dialogue domain. In this experiment, we compare different belief collection methods and different belief update methods.

Finally, we examine the Dialogue domain in Figure 5–7. Interestingly, the simple Random Collection is able to find the optimal solution fastest. We attribute this to the short planning horizon required for this task, only a few (less than five) queries are required to have a peaked belief in the user's intent. The methods which focus on reducing error (Bound Uncertainty and Error Minimization) also perform well. We see that the L_1 Norm Collection outperforms the L_1 Leaf Biased Collection. This will be examined more fully in Section 5.7. These results indicate that the MDP Heuristic Collection method does very poorly in the Dialogue domain. This is not surprising, since the optimal MDP will never guide the search in belief space towards action which are strictly information gathering, which is critical in this domain.

5.4 Size of Belief Space

In the next two sections, we investigate the effects of both varying the number of belief points collected at each iteration (N), and varying the number of iterations of belief updates (U). These parameters are used in the generic point-based method outlined in Algorithm 1 (Section 3.1). Both of these parameters can be seen as ways to alter the proportion of time spent collecting vs the time spent updating beliefs. When N is high, this shifts the balance of computation time towards belief point collection, especially when the time complexity of the update method does not scale linearly with N (as in the Perseus update method). Modifying U has the opposite effect: when we increase U, we linearly increase the time spent computing belief point updates while the collection time remains fixed. This should theoretically be useful for collection/update combinations that spend a large proportion of time collecting, compared to updating.

The size of the collected belief point set N should also be dependent on the domain characteristics, especially for the exploration-based methods such as Random, MDP Heuristic and Bound Uncertainty. It is clear that N needs to be at least as large as the number of execution steps required to reach most beliefs encountered under an optimal policy. Otherwise, the search might always terminate before encountering many useful beliefs.

In all of the following experiments, we use the Full Backup method for updates, and we vary the collection method. We chose the Full Backup method due to its overall strong performance and simplicity. In our legend, we denote the collection method used as well as the number of belief points collected at each step.

There is little surprise in the Hallway domain (Figure 5–8). The domain is simple enough that it does not matter how many belief points are collected at each step.

We present the results for the RockSample[5,5] domain in Figure 5–9. In general, there is not a drastic change in the quality of the resulting policy as we vary the number of belief points collected. The number of belief points does have a strong effect on Error Minimization, which does best with a small number of points. This is because the Error Minimization collection method is relatively slow, so it does poorly when collecting large numbers of points with few updates. We also see that some of the methods (Random, L_1 Norm) do worse with the smallest number of new

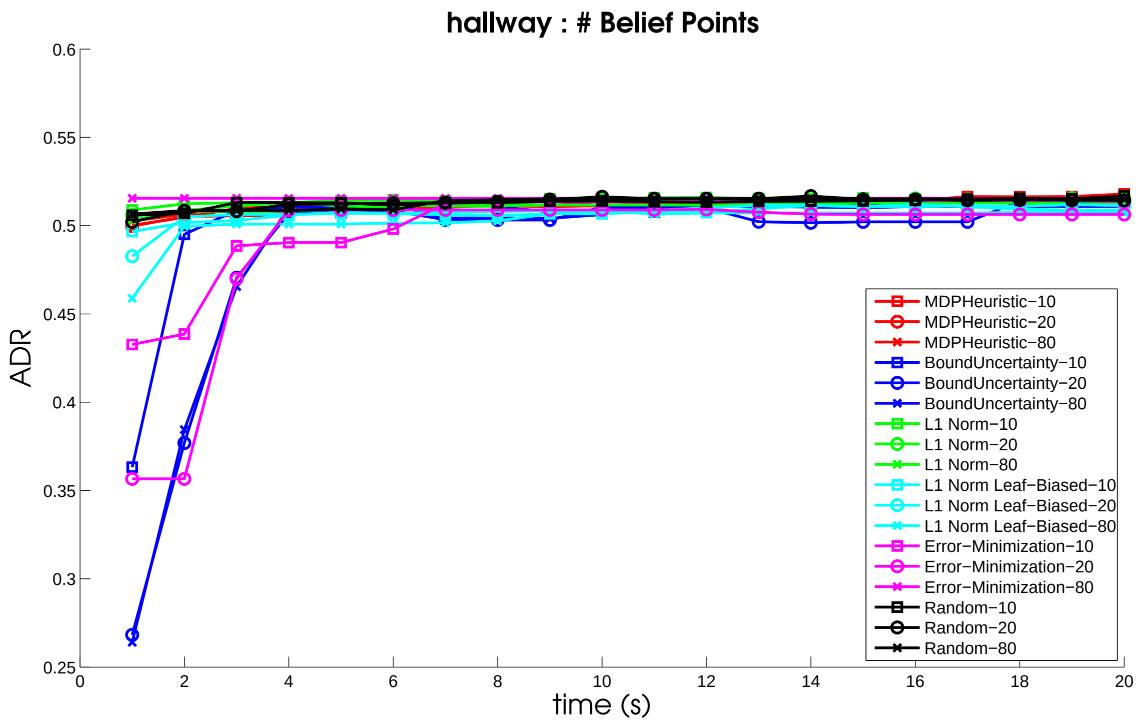

Figure 5–8: The Hallway domain. We compare the belief collection methods with the number of belief points collected per iteration.

beliefs. Since the collection is quite fast, especially for Random, they perform better when there are a larger number of points.

The results for the Underwater Navigation domain are shown in Figure 5–10. These results stress how critical it is to add more belief points at each step than the maximum possible planning horizon. Where many of the algorithms (Bound Uncertainty with all sizes of N, Random and MDP Heuristic with large sizes of N) have no problem immediately finding an optimal solution when a large number of beliefs are collected, the same methods are not able to ever find a solution when an inadequate number of points are added. Interestingly, MDP Heuristic with a small

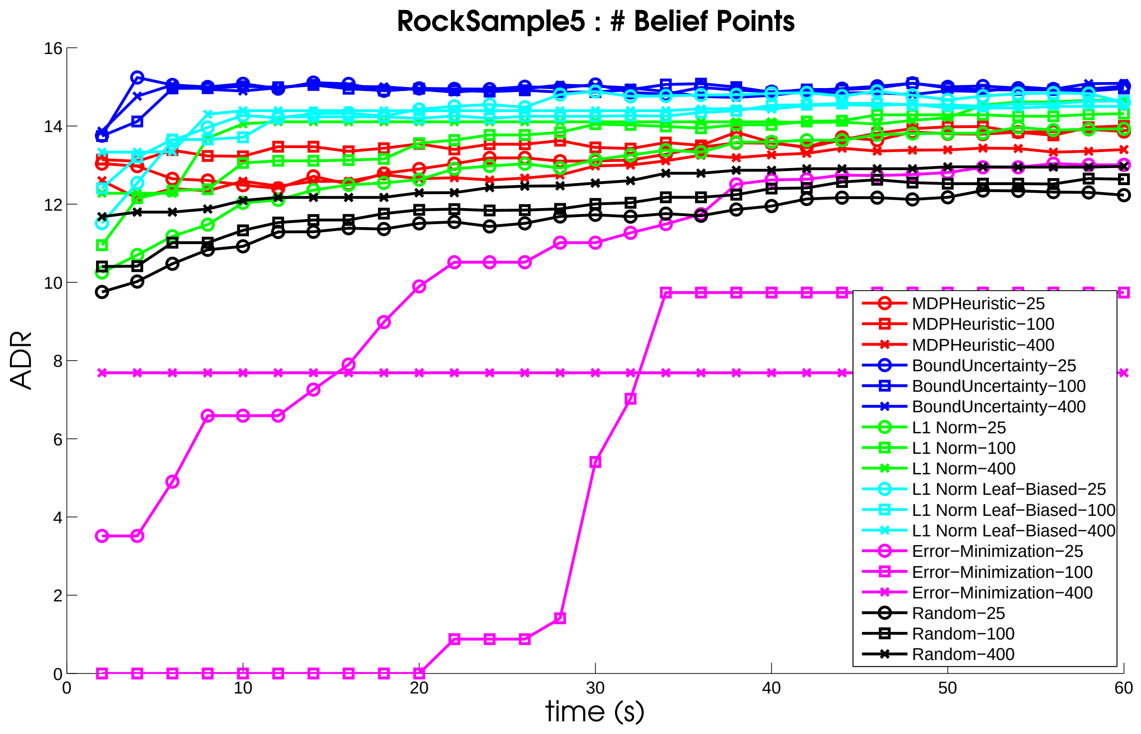

Figure 5–9: The RockSample[5,5] domain. We compare the belief collection methods with the number of belief points collected per iteration.

number of belief points is only sometimes able to find a solution. With a limited number of belief points added, only some policies are able to find the critical beliefs for an optimal policy.

In Figure 5–11, we show the results for the Dialogue domain. We see that for some of the collection methods, the choice of how many belief points to collect per iteration does have an effect. Error Minimization and L_1 Norm show much better performance with a larger number of beliefs. We could attribute this to the fact that this domain is easily solvable if the important beliefs are found (i.e. the ones reached after choosing the queries). Good performance is achieved only after these

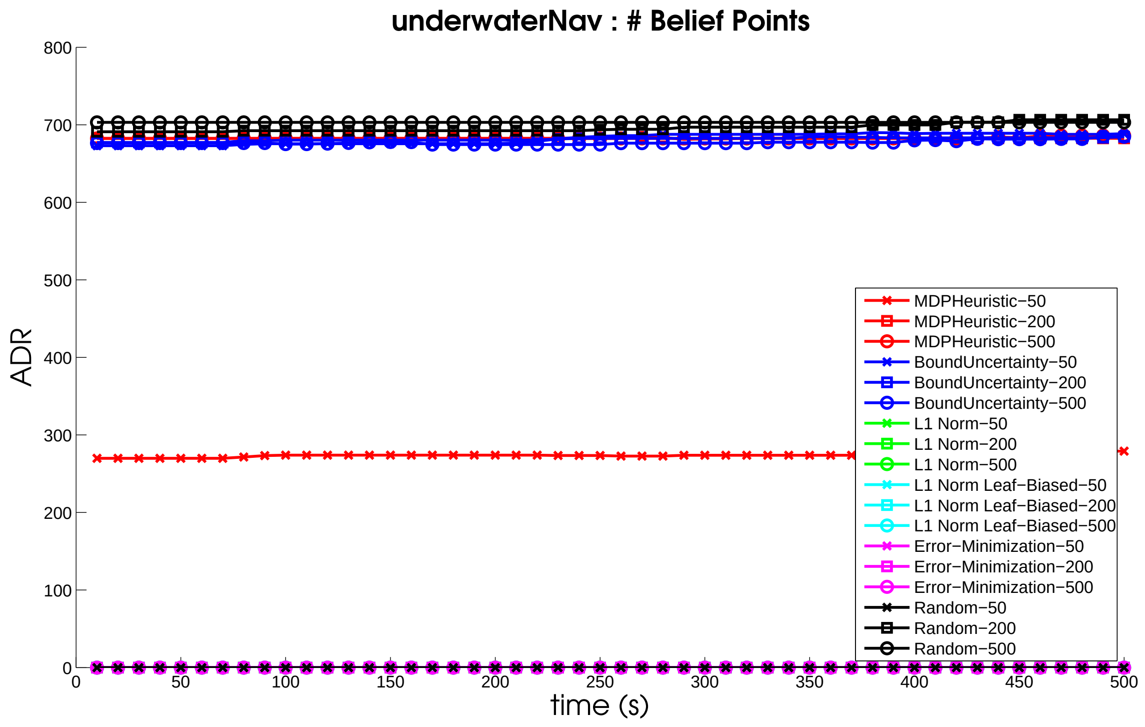

Figure 5–10: The Underwater Navigation domain. We compare the belief collection methods with the number of belief points collected per iteration.

beliefs are found. The MDP Heuristic Collection shows an opposite effect, however the resulting policies are still extremely poor, so we do not believe it to be significant.

The results for the Tag, RockSample[7,7] and FieldVisionRockSample[5,5] domains are included in Appendix A.

5.5 Iterations of Belief Point Updates

In this section, we vary the parameter U, which controls the number of iterations of belief point updates at each step in the POMDP planning, as seen in Algorithm 1. We can use this parameter to control the time spent updating beliefs versus

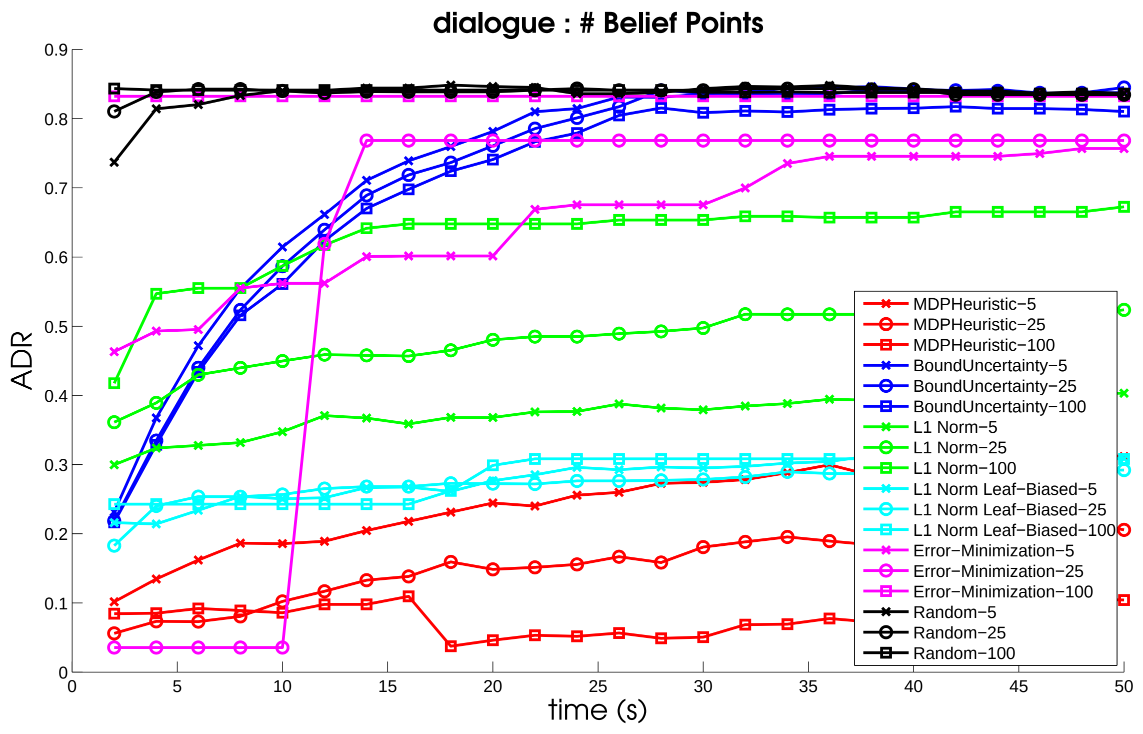

Figure 5–11: The Dialogue domain. We compare the belief collection methods with the number of belief points collected per iteration.

collecting new ones. Applying more updates should result in a better policy than a single update, however the extra time spent updating is not used for collecting new points (and updating these points instead). In theory, we should expect more updates to do relatively better when used with slower collection algorithms, since these methods should attempt to extract as good a policy as possible with the belief points they have collected.

In all cases, we use the full backup method for updates. Unless otherwise specified, the number of points collected at each step is $N = 100$ for all collection methods except Error Minimization which uses $N = 10$.

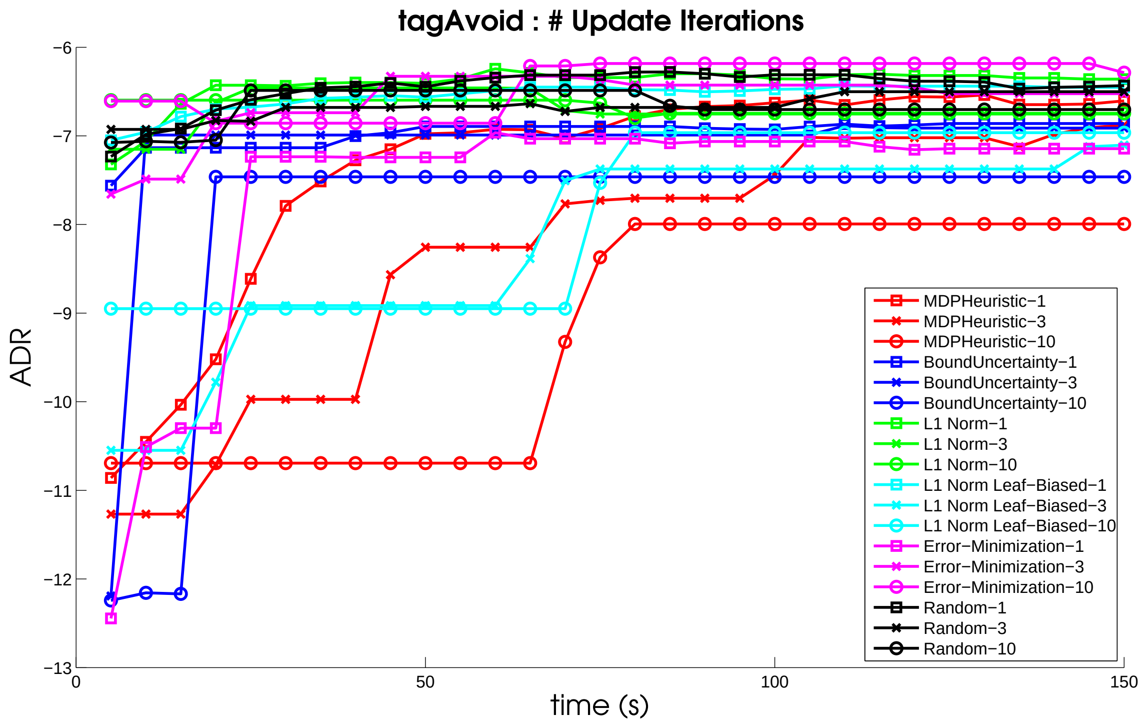

Figure 5–12: The Tag domain. We compare the belief collection methods with the number of iterations of belief point updates.

We first provide the results for the Tag domain in Figure 5–12. With many collection methods, such as Random or L_1 Norm, the number of updates executed per iteration makes little difference. However, some methods (MDP Heuristic, L_1 Leaf Biased) do slightly better with fewer updates. Since there is a set "goal" to this task, this effect may be because the algorithms spend too much time updating useless belief points.

We show the results for the RockSample[5,5] domain in Figure 5–13. For many of the collection methods (Bound Uncertainty, the L_1 Norm methods and MDP Heuristic), the number of iterations make little to no difference. In these cases, the

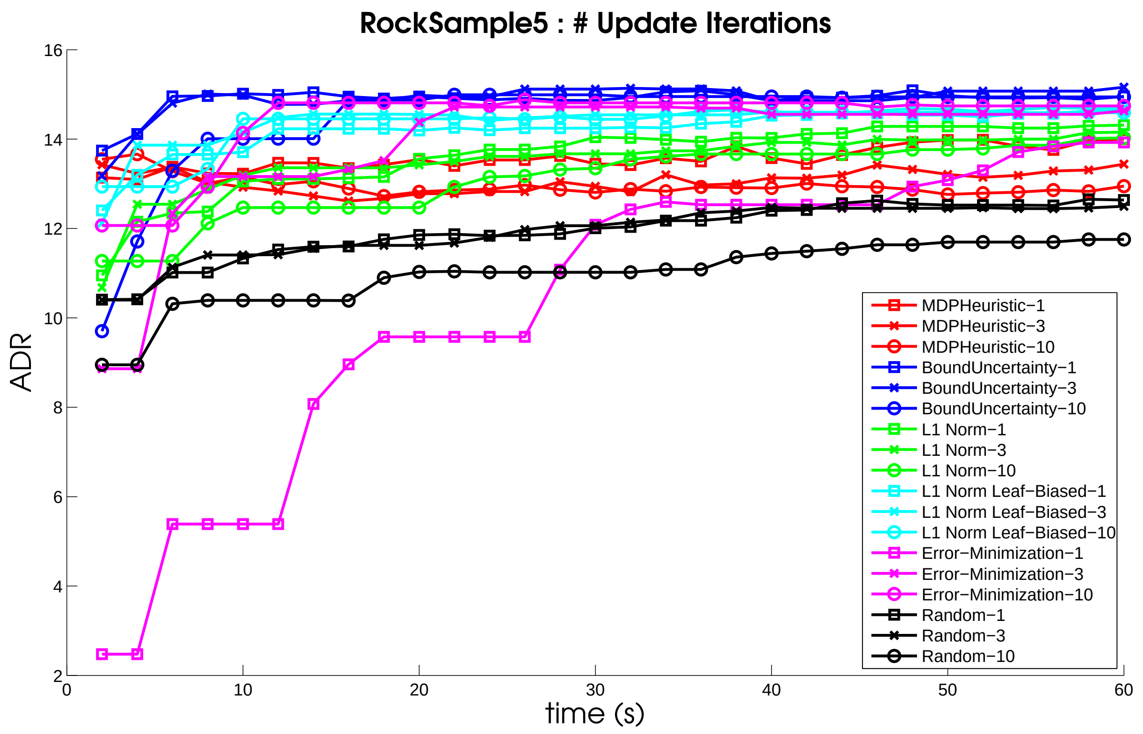

Figure 5–13: The RockSample[5,5] domain. We compare the belief collection methods with the number of iterations of belief point updates.

extra time spent updating is equally traded off by the performance increase incurred by these extra updates. However, running extra updates while using the Error Minimization collection method increases the performance in a dramatic way, with 10 updates outperforming 3 updates, which dramatically outperforms a single update. Since the Bound Uncertainty collection uses a theoretically-motivated but very expensive technique to collect beliefs, it performs much better when extra time is spent exploiting these beliefs. On the contrary, we see that Random collection performs better with a single update rather than 3 updates, which subsequently outperforms 10 updates. Since Random exploration will lead to mostly unhelpful beliefs, as well as being extremely time-efficient, it is better to only consider the belief points once and continue finding new ones. Similar results are found for RockSample[7,8] and FieldVisionRockSample[5,5], as shown in Appendix A.

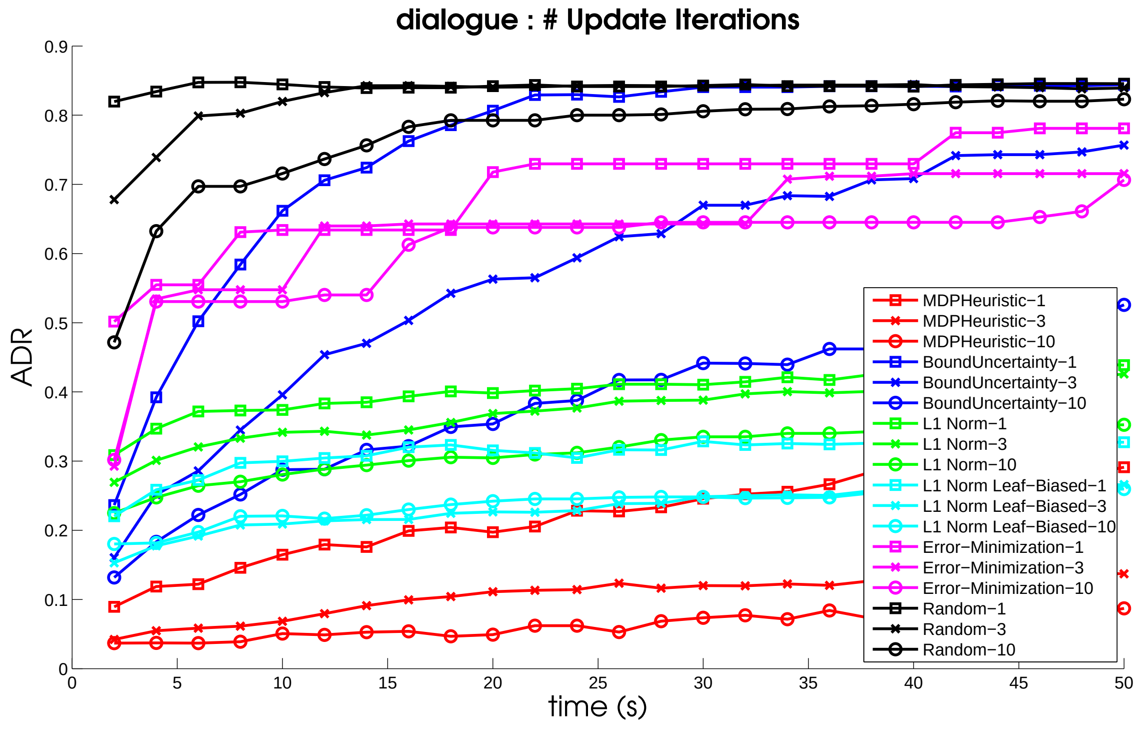

Figure 5–14: The Dialogue domain. We compare the belief collection methods with the number of iterations of belief point updates.

Figure 5–14 shows the results of varying the number of updates per iteration for the Dialogue domain. In this case, other than Error Minimization, we see a clear advantage to running fewer update iterations, especially for the Bound Uncertainty, L_1 Norm and MDP Heuristic collection methods. In this domain, once the agent has discovered the critical beliefs (by taking sequences of information gathering actions), it can quite quickly come to a solution. The difficulty in finding a good policy is in finding these critical beliefs. When the algorithms execute fewer belief point updates per iteration, it allows them more time to explore.

The results for the Hallway, RockSample[7,8], FieldVisionRockSample[5,5] and Underwater Navigation are included in Appendix A.

5.6 Confidence Intervals

In all previous graphs, we omitted confidence intervals for clarity. However, it is useful to be able to see results with the confidence intervals displayed, to give an idea of how consistent the policy is across multiple experiments. We show the results for the RockSample[5,5] and Dialogue domains in Figures 5–15 and 5–16. We show all collection methods with a single full backup update. The error bars are a single standard deviation. The RockSample[5,5] results use $N = 100$ ($N = 10$ for Error Minimization), and the results are averaged over 20 trials. The Dialogue results use $N = 10$ and are averaged over 10 trials.

The confidence intervals are in general as we would expect; the higher performing algorithms tend to have tighter error intervals, while the algorithms which are still improving have much wider errors. In the Dialogue experiment, we see that the L_1 Norm collection methods, while they perform quite poorly, have high variance. We can attribute this to the random nature of the process to select belief points as parent nodes. The MDP Heuristic collection method has a much tighter error, even with poorer performance. This can be attributed to the deterministic nature of the MDP Heuristic collection method.

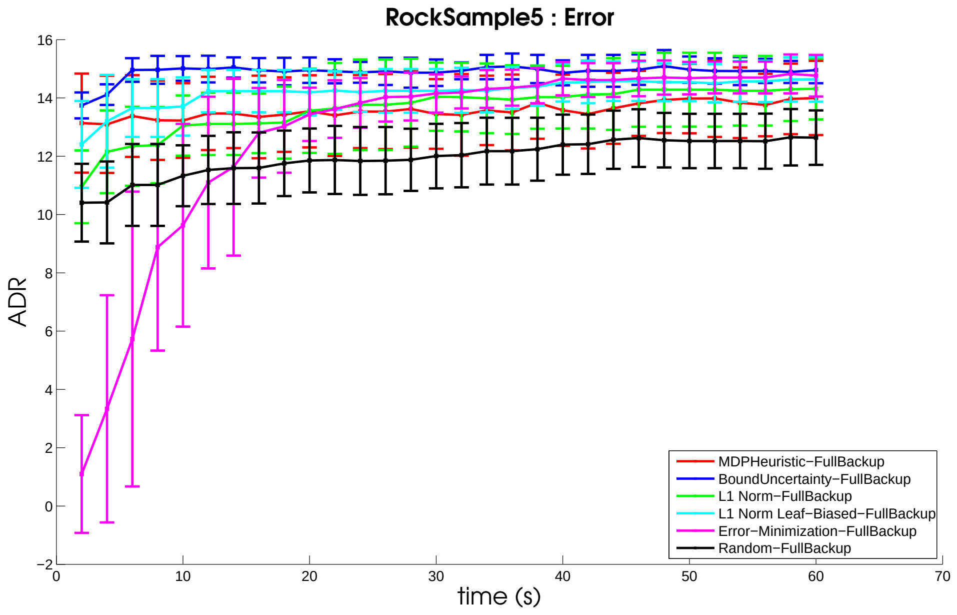

Figure 5–15: The RockSample[5,5] domain. We show the confidence intervals of the collection methods with a full belief update.

5.7 Verification of PBVI Leaf Biased

In section 4.1.4, we introduced the collection method L_1 Norm Leaf Biased, which featured two modifications intended to improve the exploration of the belief space. These modifications were a) a method of biasing the belief point selection process towards leaf nodes (beliefs with no successors), and b) a full search through all observations to pick the target farthest belief point successor.

In this section, we investigate directly the effects of these two proposed optimizations. In Figures 5–17, 5–18 and 5–19, L_1 Norm is the standard L_1 Norm collection method, L_1 Norm Full adds the full observation search when looking for successor beliefs, L_1 Norm Leaf Biased adds the random bias towards leaf belief nodes,

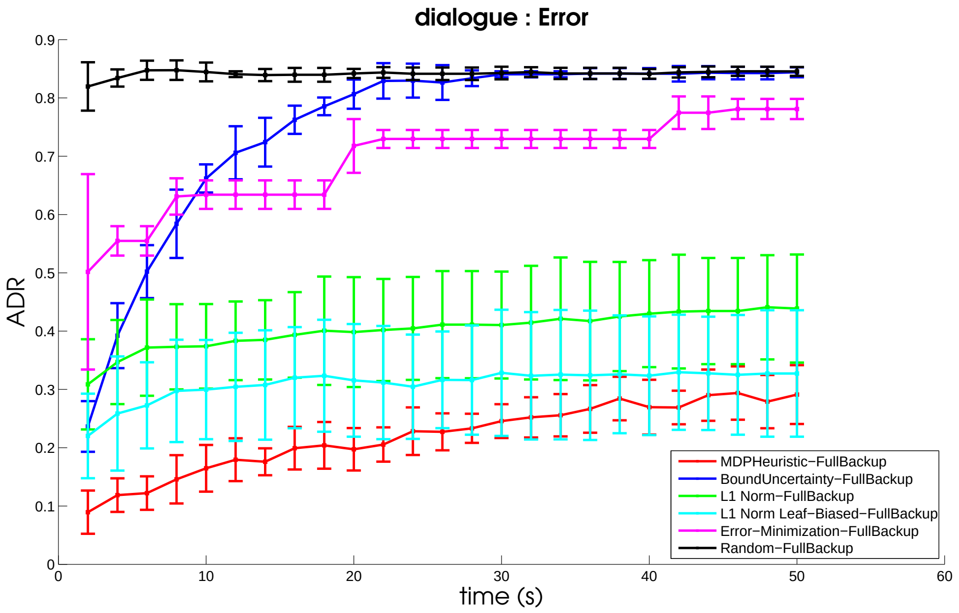

Figure 5–16: The Dialogue domain. We show the confidence intervals of the collection methods with a full belief update.

and finally L_1 Norm Full Leaf Biased adds both optimizations (as in the previous experiments).

In the RockSample[5,5] domain, shown in Figure 5–17, we see that the Leaf Biased modification gives a slight boost to the quality of the policy, while the full observation search has no effect.

The Underwater Navigation domain (shown in Figure 5–18) is much more strongly affected by the modifications than the RockSample domain. In this case, only L_1 Norm Full Leaf Biased is able to find a solution. Underwater Navigation requires a relativity long ($\sim$ 50 step) planning horizon to encounter the nearest rewards. In this type of domain, standard L_1 Norm collects many beliefs points near the initial belief, but does not build enough belief points far enough away to reach the reward.

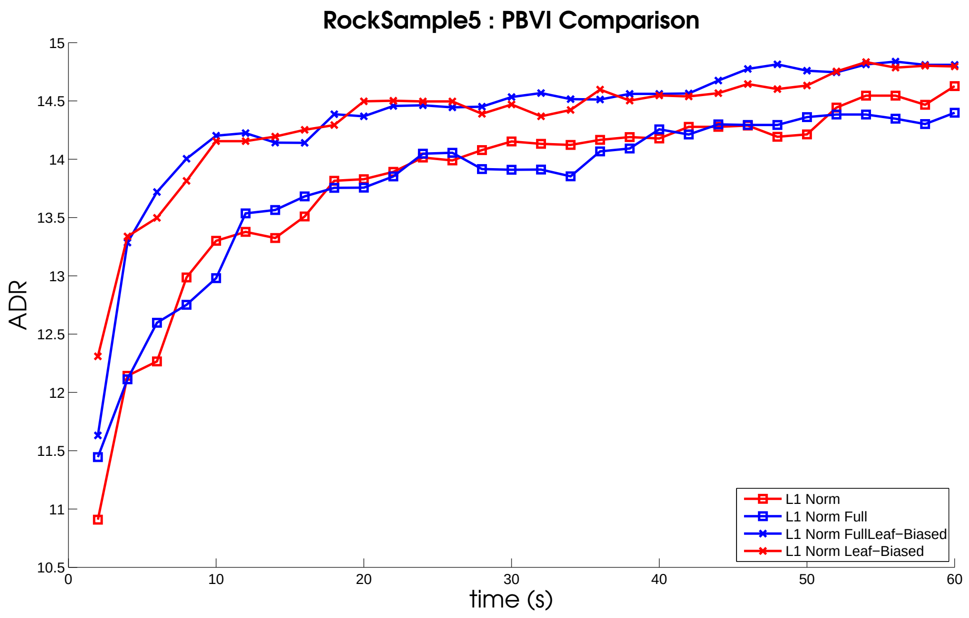

Figure 5–17: The RockSample[5,5] domain. We compare four versions of L_1 Norm Collection, with all combinations of the proposed optimizations.

We see a very different result in the Dialogue domain (Figure 5–19). In this case, L_1 Norm does best when it is not Leaf Biased. Again, this can be attributed to domain characteristics. The planning horizon for the Dialogue domain is in general very short, much shorter than most of the other domains tested. When the search for new beliefs is biased strongly towards nodes deeper in belief space, it has a negative effect on the optimal policy in the Dialogue task. This is because this bias will lead to finding beliefs through a very long planning horizon, created by executing many query actions.

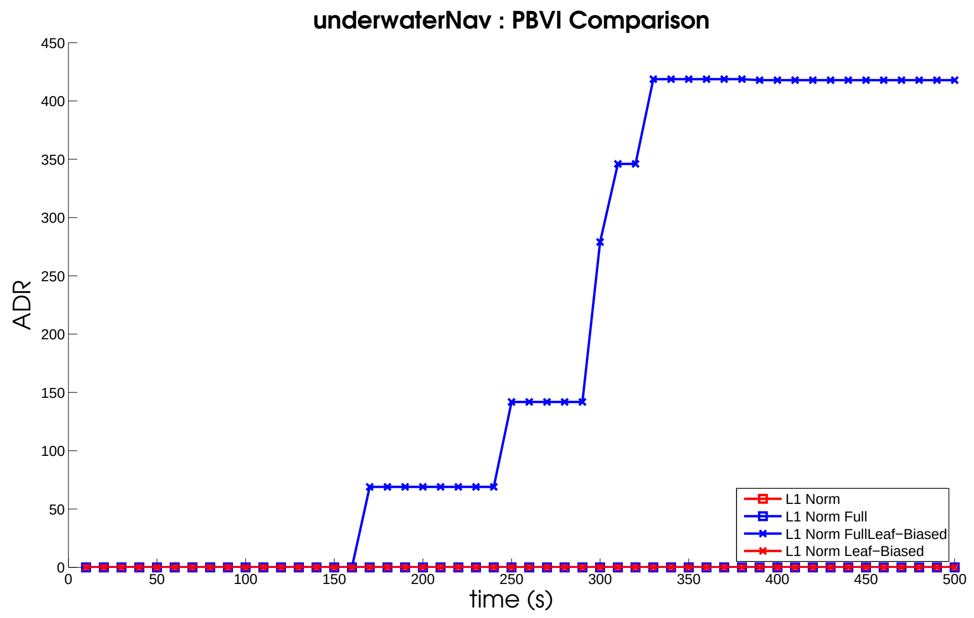

Figure 5–18: The Underwater Navigation domain. We compare four versions of PBVI, with all combinations of the proposed optimizations.

5.8 Blind Policy Lower Bound

In the updated version of HSVI [49], Smith and Simmons use an initial lower bound based on a blind policy, which was initially proposed by Hauskrecht [20], as discussed in Section 3.4. This value function approximation is created with an α-vector for each action, each associated with a policy that always takes the associated action. If initialized with the single vector lower bound in Equation 3.7, this bound is guaranteed to be tighter than the single vector lower bound.

In the following experiments, we investigate the utility of using a blind policy initialization for $V(b)$. The advantage of the blind policy is that it can give a higher

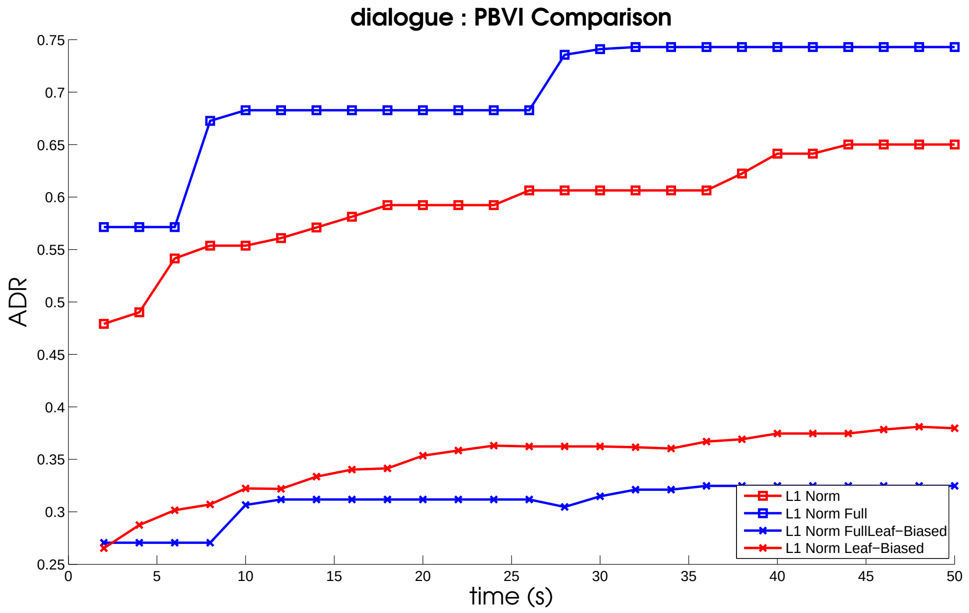

Figure 5–19: The Dialogue domain. We compare four versions of PBVI, with all combinations of the proposed optimizations.

initial value function than a single α-vector. The disadvantage is that extra computation must be done, however, it is generally minimal. We compare a selection of collection methods (L_1 Norm, Random, Bound-Uncertainty), testing each with a single-vector initialization and a blind policy initialization. The update method is always Full Backup, and the number of belief points and update iterations are the same as in the original experiments ($N = 100$ and $U = 1$).

In Figure 5–20, we see that the blind policy brings modest improvement for one collection method (Bound Uncertainty), and no difference elsewhere.

We show the results for RockSample[5,5] in Figure 5–21. In this case, we observe no advantage to using a Blind Policy for any of the collection methods.

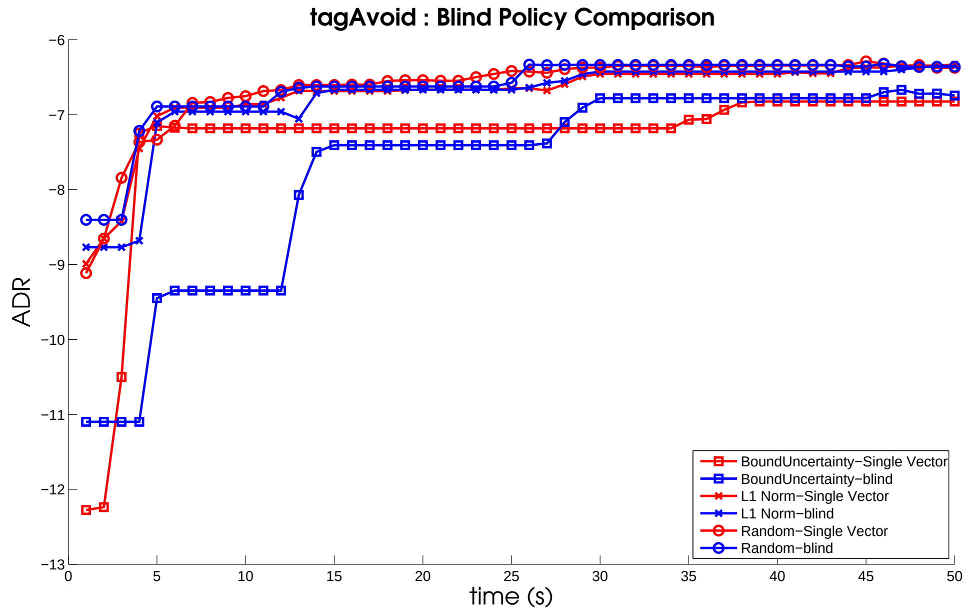

Figure 5–20: The Tag domain. We compare a selection of collection methods with either using a Single Vector value function initialization or a Blind Policy initialization.

The results shown in Figure 5–22 are much more interesting. We display the outcome of using a blind policy in the Dialogue domain. In all cases, a blind policy initialization immediately leads to a near-optimal policy, even when the method is not able to find a near-optimal strategy in the time allotted (L_1 Norm and MDP Heuristic). Recall that the Dialogue domain has three special query actions, which act only as information gathering actions. Simply applying the same query multiple times can give a very good idea of the user's intent, which results in a high performance policy. That is why an algorithm using a blind policy initialization can

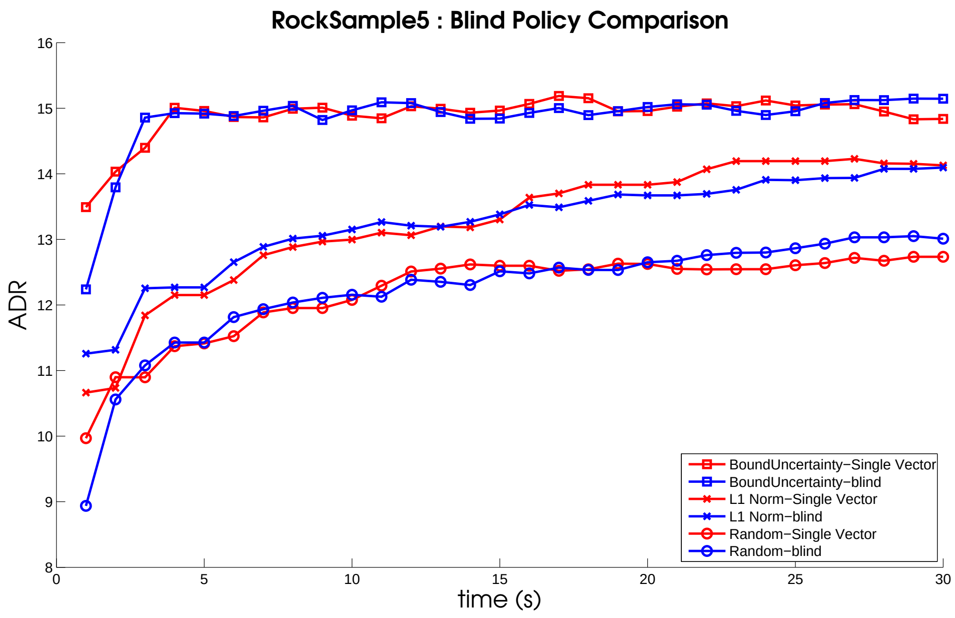

Figure 5–21: The RockSample[5,5] domain. We compare a selection of collection methods with either using a Single Vector value function initialization or a Blind Policy initialization.

succeed, whereas algorithms which use a collection method such as MDP Heuristic will perform poorly, since they do not value sequences of information gathering actions.

It is interesting to note that the blind policy initialization can lead to significant speedups, especially in a model which has been used in a real-world setting. However, we must note that this performance is solely based on the fact that repeated executions of a specific action is useful in this case. In a domain that requires a set of actions in sequence, $\{a_1, a_2, a_3\}$, to gain the same information, the blind policy initialization would provide no benefit.

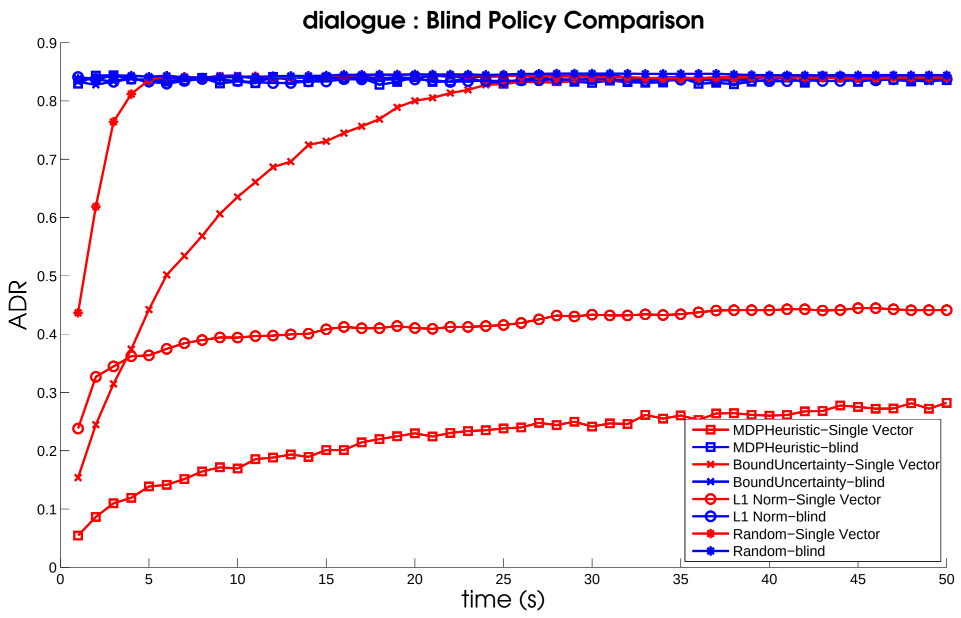

Figure 5–22: The Dialogue domain. We compare a selection of collection methods with either using a Single Vector value function initialization or a Blind Policy initialization.

5.9 Discussion

In this section, we summarize our observations of the results presented in the previous experiments. We include general conclusions about all methods tested.

Belief Point Collection Methods

One of our primary focuses in this investigation was the effect of varying the collection method used. We found that the importance of the collection method used was very domain-dependent. For example, in the Tag domain, it made little difference overall which collection method was used, while in the Underwater Navigation and Dialogue domains, the choice of collection method was critical. **Overall, the Bound**

Uncertainty collection method from HSVI [48, 49] did very well, in large part due to its use of both upper and lower bounds in its search heuristics, which can exploit information gathering actions. We also saw that **the theoretically-motivated Error Minimization method (from PEMA [38]) performs much slower than most other collection methods, although it is able to do well when the domain has a short planning horizon and requires information gathering** actions as in the Dialogue domain. When coupled with multiple rounds of belief point updates, we found that Error Minimization was able to perform well on a variety of domains. **The MDP Heuristic [46] collection method also does very well overall. However, it performs poorly when there are specific information gathering actions** (as in Dialogue) since the MDP solution places no value on information. **The L_1 Norm collection method from PBVI [39] does not do as well as the more optimistic approaches, and had difficulty solving methods with a very long planning horizon**, which we intended to amend with the modifications used in L_1 Norm Leaf Biased. Interestingly, Random collection (used in Perseus [52]) is able to do quite well in many of the domains, due to the speed of the collection and the fact that many of the domains have critical beliefs that may be far, but are easy to find. The Random Collection method performed poorly in the RockSample domains, where the critical beliefs were more difficult to find, requiring executing the right "Check" actions near the associated rock. As well, Random was in fact the fastest algorithm for the Dialogue domain, which might imply that there is still room to improve the recent belief point collection methods, especially when information gathering actions are concerned. However, this must be investigated since the required planning horizon is very short.

Analysis of L_1 Norm Leaf Biased Collection

In this work, we proposed two modifications for the L_1 Norm collection method introduced in PBVI. The first is related to collecting a wider sample of belief points by iterating over observations, and the second deals with biasing the belief point selection to belief nodes that are leaves in the exploration tree. In these experiments, we saw that the effect of these modifications on the overall policy is dependent on the domain characteristics. **The optimizations of the L_1 Norm Leaf Biased Collection method seem to provide a boost when the required policy horizon is long**, such as in a long navigation task (Underwater Navigation). However, **the modifications can hurt performance when the planning horizon is short** as in the Dialogue domain.

Belief Point Update Methods

A second major focus in this work was the effect of varying which belief points were updated at each iteration. Other than for Error Minimization (which favored a full backup due to speed concerns), **the effect of the belief update method was usually small, and did not favour a specific method**. We did not find much of an advantage to using the Perseus-style backups, especially when used with the heuristic methods like Bound Uncertainty and MDP Heuristic.

Number of Collected Belief Points

We investigated the effect of varying the N parameter, the number of belief points added at each iteration. Overall, **there was not much of an effect when varying the number of belief points collected**. It is clear that it is critical to consider N to be at least as long as the longest planning horizon required, especially for the methods which collect beliefs through traces (Random, Bound Uncertainty, MDP Heuristic). In the original algorithms which use these methods, they collect belief points until the trial ends, which ends the collection iteration. Our results show that this method is a reasonable approach.

Number of Belief Update Iterations

We also investigated the effect of varying the U parameter, which is the number of iterations of belief updates executed per step in the POMDP solving. In general, we saw that **a single round of updates is generally sufficient**, and a single round generally performed at least as well as 3 or 10 rounds of updates. The only notable exception is Error Minimization, which performed much better with multiple updates.

Blind Policy Initialization

Finally, we investigated the utility of initializing the value function with the result of a blind policy iteration. We found that, in most domains, there was no effect. However, in Dialogue, it was able to give a huge performance boost, since a blind policy in Dialogue is able to explore the key beliefs found when executing the information gathering “Query” actions. We would therefore **recommend initializing with a blind policy**, since it is very cheap to compute, but the utility of a blind policy is dependent on domain characteristics.

5.10 Future Work

In this chapter, we detailed a series of experiments in which we investigated the effect of varying the collection and updating methods of POMDP solvers, as well as measuring the effect of different algorithmic parameters. However, many elements of point-based solvers remain to be investigated. In this section we propose further experiments that would have supplemented our results to date.

Throughout our experiments, we kept the α-vector pruning fixed, as described in Section 4.2.4. This includes the immediate pruning step used in Equation 4.4, as well as single-vector domination checks. Informally, we have seen that the choice of pruning techniques used can have a strong effect on the speed of the POMDP solvers. The choice of pruning techniques should be more deeply investigated.

In several of the domains used in this work, especially the RockSample and Tag domains, even at the maximum alloted planning time, the ADR is not uniform across the different algorithms. This suggests that not all algorithms have converged. Many of the collection algorithms have theoretical results bounding the error on the value function compared to the optimal value function, and can converge to the optimal policy in the long term. Algorithms with such optimality guarantees include Bound Uncertainty [48,49], L_1 Norm [39] and Error Minimization [38]. Even Random collection will eventually cover the entire belief space, although of course there are no worst-case bounds on how long that would take. Since the MDP Heuristic collection method has no way to deal with strictly information gathering actions, it is not guaranteed to converge for all domains. A further investigation of the relationship between these convergence guarantees and practical results would be useful.

Our investigations covered a wide range of domains. We saw that the specific domain properties played a large role in how effective the planning algorithms would be, such as the long planning horizon of Underwater Navigation, the scalable size and non-goal nature of the RockSample domains and the information gathering actions of Dialogue. However, most of the domains examined are modifications of the classic tabular navigation domains. As well, even the domain with information gathering actions, Dialogue, has an extremely simple model for gathering information, which as we saw was solvable by a blind policy. We require more complicated problems for decision making, which satisfy some of the following properties. First, we require testing on domains with a higher value of information, in the sense of requiring more difficult strategies to gather information, i.e. the agent is required to plan to successfully retrieve information. Second, we require domains with much more challenging partial observability. Many of the domains used have quite simple models of partial observability, which makes the job of the planner much easier. A domain with substantial, long-term partial observability would be a useful benchmark for new POMDP algorithms.

We discussed the SARSOP algorithm proposed by Kurniawati et al. [26] in Section 3.7. This algorithm has shown promising empirical results compared to other point-based POMDP solvers. The collection method used in SARSOP was unfortunately omitted from these results due to our goal of testing on a single software platform, which did not include the SARSOP methods. A future empirical analysis of point-based methods should include the SARSOP collection method.

In this work, we focused on the computation time requirements for POMDP solvers, but the memory requirements for POMDP planning can be quite steep. Additional optimizations can be made by using more memory, such as caching all belief points explored. Memory requirements can become an issue especially for deployment of POMDP solvers onboard autonomous agents which use embedded systems with smaller amounts of memory. Results illustrating the memory requirements of point-based POMDP solvers, especially in terms of optimizations used, would be useful for determining the best POMDP planning algorithms for embedded systems.

CHAPTER 6

Conclusion

Planning under uncertainty is still an unsolved problem in modern AI research. Planning has a range of applications: in medical systems, scheduling systems, robotics and very likely many new fields in the future. In all domains, it is critical that the planning algorithms are robust, scalable, and able to make useful decisions in the face of deep uncertainties.

In this work, we have discussed the Partially Observable Markov Decision Process (POMDP) framework, which offers a rich and flexible model for decision making under uncertainty. Our focus is on the class of approximate POMDP solvers known as point-based POMDP solvers, which plan using a subset of belief points in the POMDP belief space. We provided a survey on modern point-based approaches, and focused on comparing their associated belief point collection and belief point update methods, as well as investigating the effect of key parameters on the quality of the resulting policies. Our goal is to provide a comprehensive view of the state of the art in the point-based POMDP planning field, as well as provide new insights into how to structure point-based algorithms. We provided our overall conclusions in regards to strengths and weaknesses of these methods in Section 5.9. Conducting such empirical studies has side benefits, as they can often lead to novel approaches. For example, we developed the L_1 Norm Leaf Biased collection method as part of our investigation into why the standard L_1 Norm collection method was unable to find a solution to the Underwater Navigation domain in the time periods tested.

We have also highlighted future directions for expanding this analysis of point-based POMDP solvers, including testing pruning approaches, examining memory constraints, extending the set of algorithms tested and expanding to more benchmark domains. We stressed the importance of using a standardized code-base for comparing algorithms, since differences in the implementation of the algorithms (programming language, optimizations used) can lead to biases in the results. We hope that future belief collection methods and other modifications to the standard point-based POMDP planner can be integrated into our test system and be compared fairly with other algorithms.

The development of point-based POMDP solving algorithms has drastically increased the efficiency of the state of the art in POMDP planners. The earlier solvers were generally tested on problems with at most dozens of states, but the advances made in the point-based methods have allowed POMDPs with tens of thousands of states to be solvable. However, the current generation of point-based solvers are still not able to scale up to large real world domains. There have been several proposed methods to help scale POMDPs to larger domains. Factored representations [13] are able to more compactly represent the state space of the model. Many POMDPs have several independent components to their state space, for example, the location of the robot and status of the rocks in the RockSample domain. Factored approaches exploit the independence of these structures of POMDPs to greatly speed up computation. Online approaches [44], which provide addition planning during real-time execution, are also able to greatly improve policies in partially observable environments. These techniques, as well as using improved point-based approaches, will help us apply POMDPs to a greater range of real world applications.

Appendix A

Here we present the results of experiments that were omitted from Chapter 5.

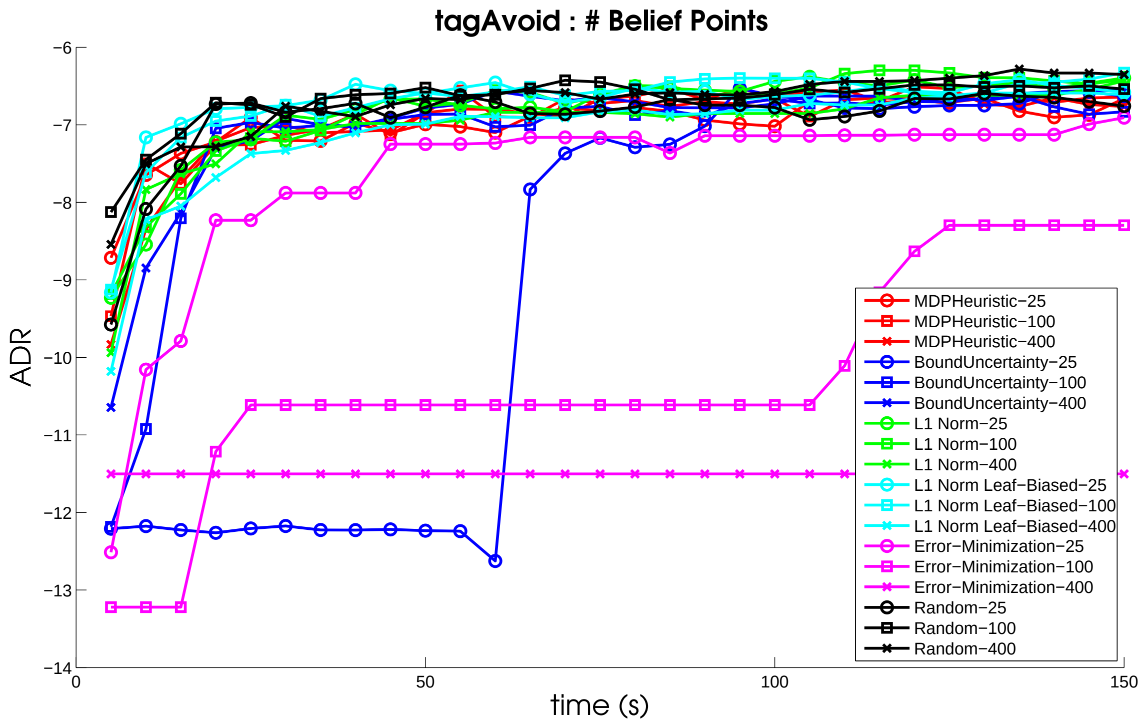

Figure 6–1: The Tag domain. We compare the belief collection methods with the number of belief points collected per iteration.

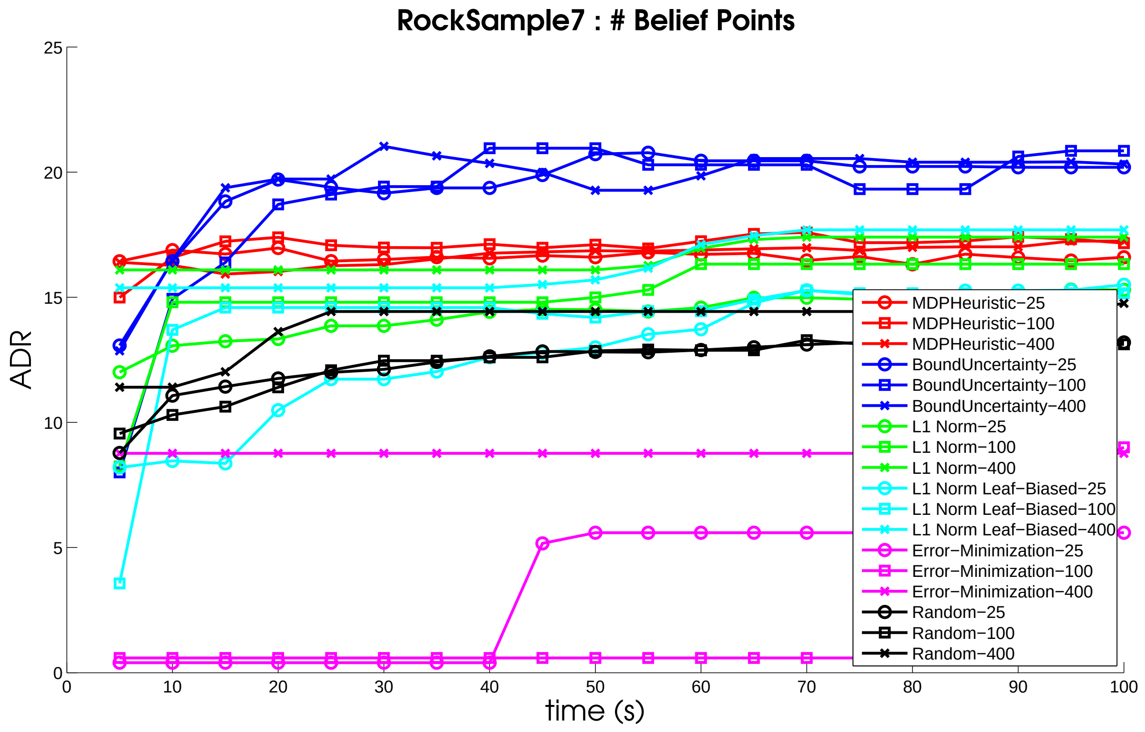

Figure 6–2: The RockSample[7,8] domain. We compare the belief collection methods with the number of belief points collected per iteration.

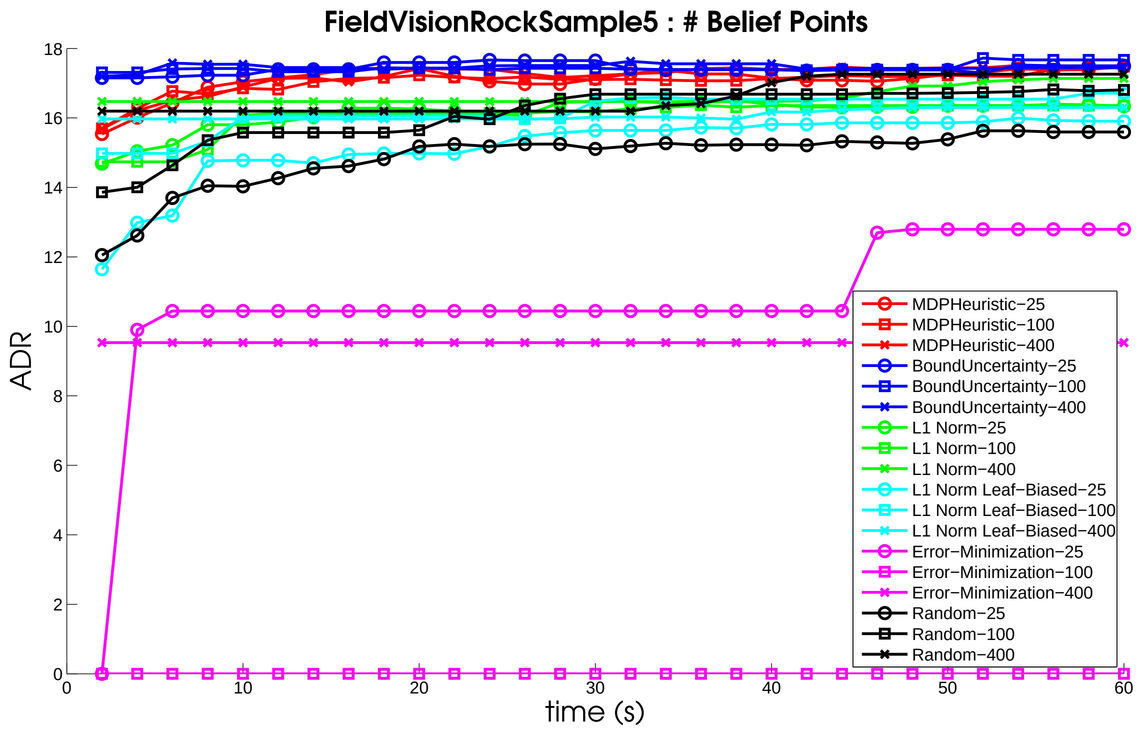

Figure 6–3: The FieldVisionRockSample[5,5] domain. We compare the belief collection methods with the number of belief points collected per iteration.

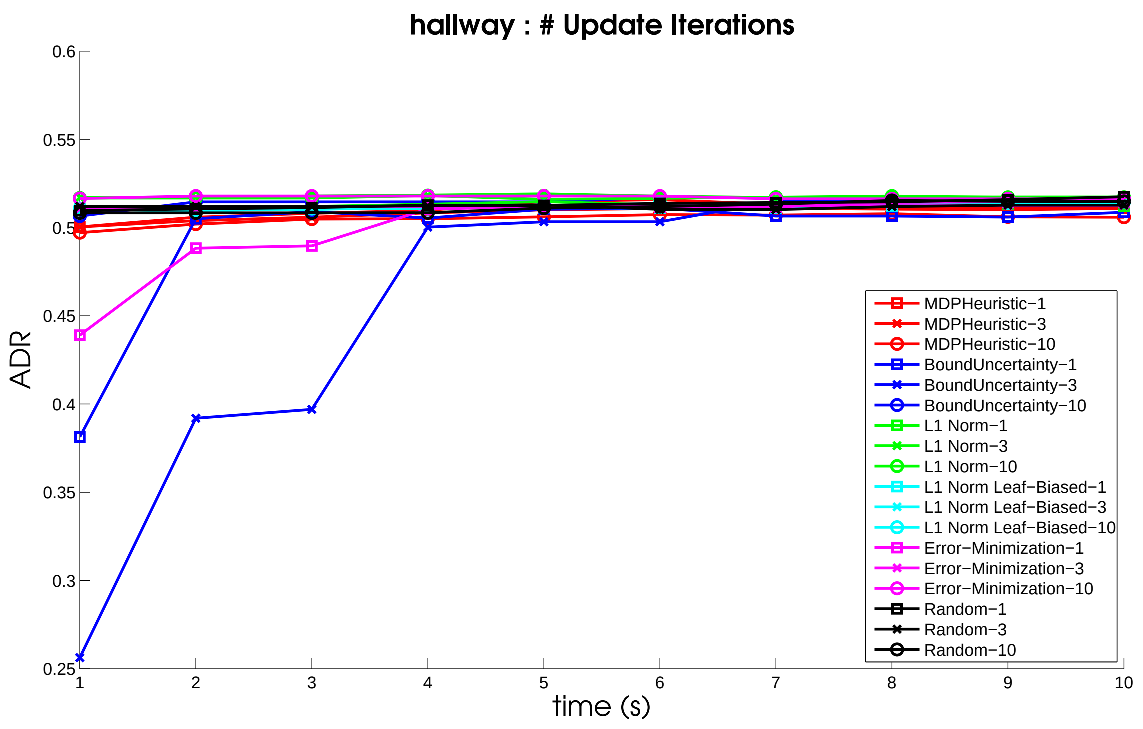

Figure 6–4: The Hallway domain. We compare the belief collection methods with the number of iterations of belief point updates.

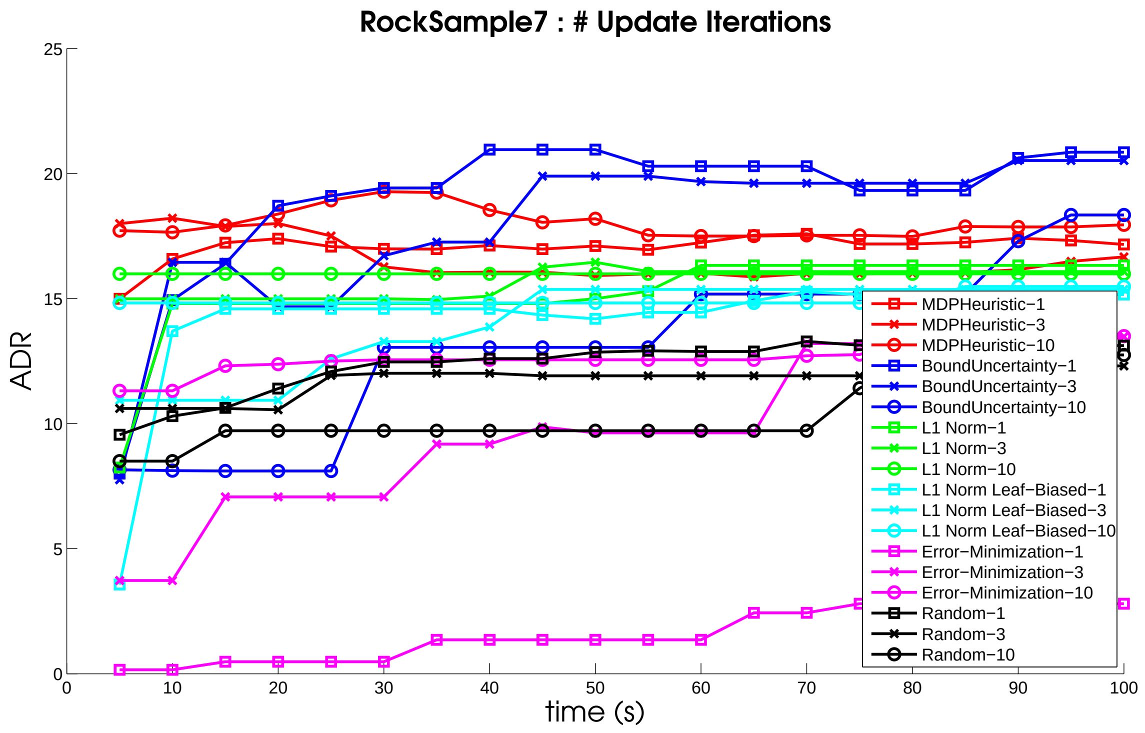

Figure 6–5: The RockSample[7,8] domain. We compare the belief collection methods with the number of iterations of belief point updates.

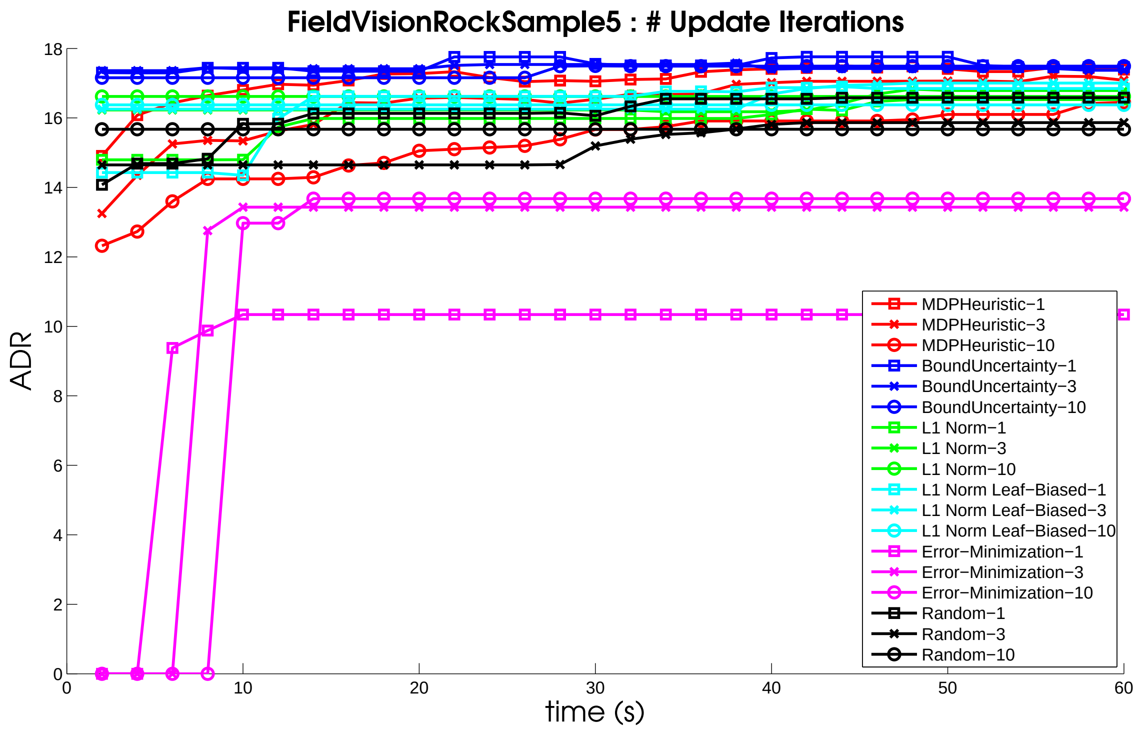

Figure 6–6: The FieldVisionRockSample[5,5] domain. We compare the belief collection methods with the number of iterations of belief point updates.

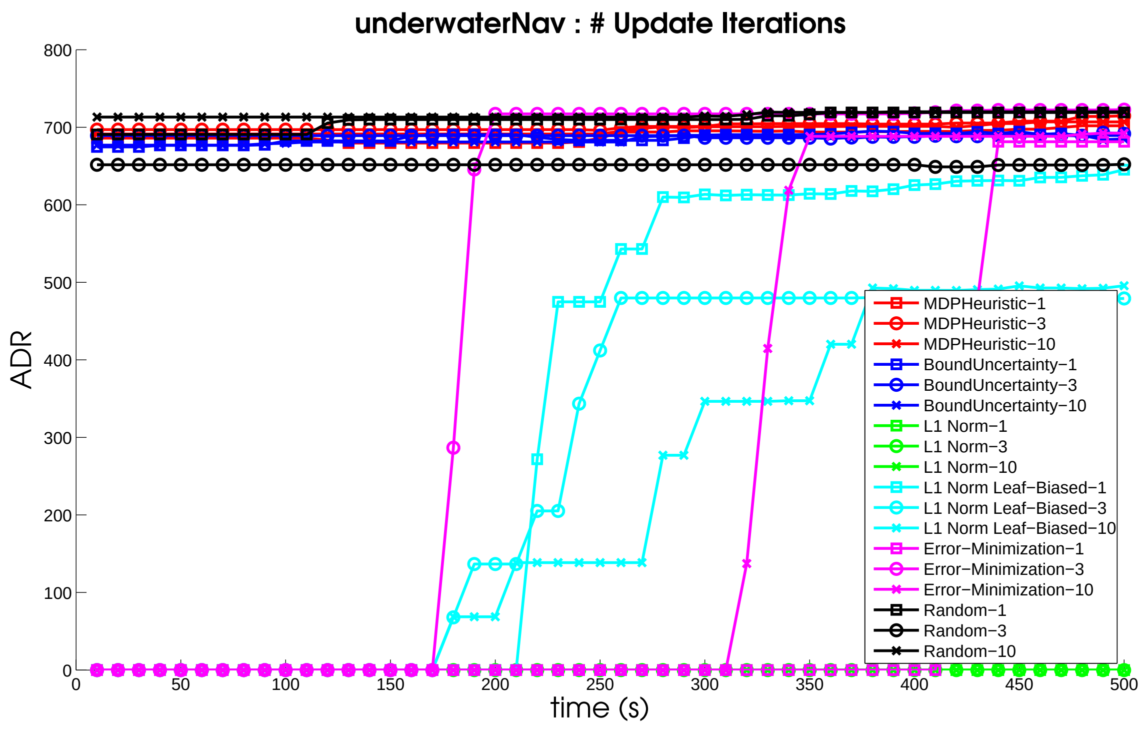

Figure 6–7: The Underwater Navigation domain. We compare the belief collection methods with the number of iterations of belief point updates.

References

[1] `http://www.cs.mcgill.ca/~smartwheeler/dialogue-feb10.POMDP`.

[2] J. Allen, J. Hendler, and A. Tate. *Readings in Planning*. Morgan Kaufmann, 1990.

[3] K. J. Astrom. Optimal control of Markov decision processes with incomplete state estimation. *Journal of Mathematical Analysis and Applications*, 10:174–205, 1965.

[4] A. Atrash, R. Kaplow, J. Villemure, R. West, H. Yamani, and J. Pineau. Development and validation of a robust speech interface for improved human-robot interaction. *International Journal of Social Robotics*, 1:345–356, 2009.

[5] J. A. Bagnell, S. Kakade, A. Ng, and J. Schneider. Policy search by dynamic programming. In *Neural Information Processing Systems (NIPS)*, volume 16, December 2003.

[6] L. C. Baird and A. W. Moore. Gradient descent for general reinforcement learning. In *Neural Information Processing Systems (NIPS)*, pages 968–974, 1998.

[7] J. Baxter and P. L. Bartlett. Reinforcement learning in POMDP's via direct gradient ascent. In *International Conference on Machine Learning (ICML)*, pages 41–48, 2000.

[8] J. Baxter, P. L. Bartlett, and L. Weaver. Experiments with infinite-horizon, policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:351–381, 2001.

[9] R. Bellman. Dynamic programming. In *Princeton University Press*, 1957.

[10] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. Technical Report CMU-CS-95-221, Carnegie Mellon University, Dec. 1995.

[11] B. Bonet. An epsilon-optimal grid-based algorithm for partially observable Markov decision processes. In *International Conference on Machine Learning (ICML)*, pages 51–58, 2002.

[12] C. Boutilier, T. Dean, and S. Hanks. Decision theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[13] C. Boutilier and D. Poole. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1168–1175, 1996.

[14] A. Cassandra, M. L. Littman, and N. L. Zhang. Incremental Pruning: A simple, fast, exact method for partially observable Markov decision processes. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 54–61, San Francisco, CA, 1997. Morgan Kaufmann Publishers.

[15] E. M. Clarke and O. Grumberg. Research on automatic verification of finite-state concurrent systems. Technical Report CMU-CS-87-105, CMU, 1987.

[16] Z. Feng and S. Zilberstein. Efficient maximization in solving POMDPs. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 975–980, 2005.

[17] R. Fikes, P. E. Hart, and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving. *Artificial Intelligence*, 2:189–208, 1971.

[18] H. Geffner and B. Bonet. Solving large POMDPs using real time dynamic programming. In *Proceedings AAAI Fall Symp. on POMDPs*, 1998.

[19] E. Hansen. Solving POMDPs by searching in policy space. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 211–219, 1998.

[20] M. Hauskrecht. Incremental methods for computing bounds in partially observable Markov decision processes. In *AAAI/IAAI*, pages 734–739, 1997.

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

[22] R. A. Howard. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, Massachusetts, 1960.

[23] A. Jazwinsky. *Stochastic Processes and Filtering Theory.* Academic Press, New York, 1970.

[24] L. Kaelbling. *Learning in Embedded Systems.* The MIT Press: Cambridge, MA, 1993.

[25] L. Kaelbling, M. Littman, and A. Cassandra. Planning and acting in partially observable stochastic domains. In *Artificial Intelligence*, pages 99–134, 1998.

[26] H. Kurniawati, D. Hsu, and W. Lee. SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces. In *Proc. Robotics: Science and Systems*, 2008.

[27] M. L. Littman. *Algorithms for Sequential Decision Making.* PhD thesis, Department of Computer Science, Brown University, Providence, RI, Feb. 1996. Also Technical Report CS-96-09.

[28] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling. Learning policies for partially observable environments: Scaling up. In *International Conference on Machine Learning (ICML)*, pages 362–370, 1995.

[29] W. S. Lovejoy. Computationally feasible bounds for partially observed Markov decision processes. *Operations Research*, 39(1):162–175, Jan.–Feb. 1991.

[30] W. S. Lovejoy. A survey of algorithmic methods for partially observable Markov decision processes. *Annals of Operations Research*, 28(1):47–65, 1991.

[31] D. A. McAllester and S. Singh. Approximate planning for factored POMDPs using belief state simplification. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 409–416, 1999.

[32] N. Meuleau, K.-E. Kim, L. Kaelbling, and A. R. Cassandra. Solving POMDPs by searching the space of finite policies. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 417–426, 1999.

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

[34] A. Y. Ng, R. Parr, and D. Koller. Policy search via density estimation. In *Neural Information Processing Systems (NIPS)*, pages 1022–1028, 1999.

[35] I. R. Nourbakhsh, R. Powers, and S. Birchfield. DERVISH - an office-navigating robot. *AI Magazine*, 16(2):53–60, 1995.

[36] S. Paquet, L. Tobin, and B. Chaib-draa. An online POMDP algorithm for complex multiagent environments. In *4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 970–977, 2005.

[37] J. S. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Knowledge Representation and Reasoning*, pages 103–114, 1992.

[38] J. Pineau and G. Gordon. POMDP planning for robust robot control. In *International Symposium on Robotics Research (ISRR)*, volume 28, pages 69–82. Springer, 2005.

[39] J. Pineau, G. Gordon, and S. Thrun. Point-based value iteration: An anytime algorithm for POMDPs. In *International Joint Conference on Artificial Intelligence*, pages 1025–1032, 2003.

[40] D. Poole, A. Mackworth, and R. Goebel. *Computational Intelligence: A Logical Approach.* Oxford University Press, Oxford, 1998.

[41] P. Poupart and C. Boutilier. Bounded finite state controllers. In *Neural Information Processing Systems (NIPS)*, 2003.

[42] S. Ross and B. Chaib-draa. AEMS: An anytime online search algorithm for approximate policy refinement in large POMDPs. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2592–2598, 2007.

[43] S. Ross, B. Chaib-draa, and J. Pineau. Bayesian reinforcement learning in continuous POMDPs with application to robot navigation. In *Proceedings of the 2001 IEEE International Conference on Robotics & Automation (ICRA)*, pages 2845–2851, 2008.

[44] S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa. Online planning algorithms for POMDPs. *Journal of Artificial Intelligence Research (JAIR)*, 32:663–704, 2008.

[45] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach. Third Edition.* Prentice Hall, 1995.

[46] G. Shani, R. Brafman, and S. Shimony. Forward search value iteration for POMDPs. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2007.

[47] R. D. Smallwood and E. J. Sondik. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, 21:1071–1088, 1973.

[48] T. Smith and R. Simmons. Heuristic search value iteration for POMDPs. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2004.

[49] T. Smith and R. G. Simmons. Point-based POMDP algorithms: Improved analysis and implementation. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 542–547, 2005.

[50] Soderland and Weld. Evaluating nonlinear planning. Technical Report 91-02-03, University of Washington, 1991.

[51] E. Sondik. *The Optimal Control of Partially Observable Markov Decision Processes*. PhD thesis, Stanford University, 1971.

[52] M. Spaan and N. Vlassis. Perseus: Randomized point-based value iteration for POMDPs. In *Journal of Artificial Intelligence Research*, pages 195–220, 2005.

[53] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[54] C. Szepesvari. Reinforcement learning algorithms for MDPs - a survey. Technical Report TR09-13, University Of Alberta, 2009.

[55] R. Washington. BI-POMDP: Bounded, incremental, partially-observable Markov-model planning. *Lecture Notes in Computer Science*, 1348:440–451, 1997.

[56] D. S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.

[57] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[58] N. L. Zhang and W. Liu. Planning in stochastic domains: Problem characteristics and approximation. Technical Report HKUST-CS96-31, Department of Computer Science, Hong Kong University of Science and Technology, 1996.

[59] N. L. Zhang and W. Zhang. Speeding up the convergence of value iteration in partially observable Markov decision processes. *Journal of Artificial Intelligence Research (JAIR)*, 14:29–51, 2001.

[60] R. Zhou and A. Hansen. An improved grid-based approximation algorithm for POMDPs. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 707–716, 2001.

KEY TO ABBREVIATIONS

FIB:	Fast Informed Bound
FSM:	Finite State Machine
FSVI:	Forward Search Value Iteration
HSVI:	Heuristic Search Value Iteration
MDP:	Markov Decision Process
MLS:	Most Likely State
PBVI:	Point-Based Value Iteration
PEMA:	Point-based Error Minimization Algorithm
POMDP:	Partially Observable Markov Decision Process
SARSOP:	Successive Approximations of the Reachable Space under Optimal Policies