Efficient Model Learning for Dialog Management

by

Finale Doshi

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2007

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 21, 2007

Certified by . . . . . .
Nicholas Roy
Assistant Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Efficient Model Learning for Dialog Management

by

Finale Doshi

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2007, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Partially Observable Markov Decision Processes (POMDPs) have succeeded in many planning domains because they can optimally trade between actions that will increase an agent's knowledge about its environment and actions that will increase an agent's reward. However, POMDPs are defined with a large number of parameters which are difficult to specify from domain knowledge, and gathering enough data to specify the parameters a priori may be expensive. This work develops several efficient algorithms for learning the POMDP parameters online and demonstrates them on a dialog manager for a robotic wheelchair. In particular, we show how a combination of specialized queries ("meta-actions") can enable us to create a robust dialog manager that avoids the pitfalls in other POMDP-learning approaches. The dialog manager's ability to reason about its uncertainty—and take advantage of low-risk opportunities to reduce that uncertainty—leads to more robust policy learning.

Thesis Supervisor: Nicholas Roy
Title: Assistant Professor of Aeronautics and Astronautics

Acknowledgments

I would like to thank my advisor Nicholas Roy for his support, suggestions, and sense of humor. I also would like to thank my friends and labmates for their helpful discussions, contributions to our robotic wheelchair, and patience as subjects and listeners. Last but not least, I would like to thank my parents, for their moral support, and the NDSEG Fellowship, for its financial support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Spoken language interfaces provide a natural way for humans to interact with robots. In an ideal setting, people would not be burdened by having to recall specific keywords, nor would they have to drop whatever they were doing to type into keyboard: they would be able to use natural phrases to command systems. Over time, the system would adapt to its user's speaking style and preferences. Such a system could be especially useful for people unable to use a keyboard interface, such as those with limited mobility. However, even if we ignore natural language processing and voice recognition—separate fields unto themselves—we still have the fundamental question of dialog management, that is, how the system should behave given some set of user inputs. In this work, we develop and explore several approaches for adaptable dialog management targeted to a robotic wheelchair.

Uncertainty in Dialog Management. One challenge for a dialog management system is noisy voice recognition. Depending on the person, the system may confuse similar sounding words (such as "copy machine" and "coffee machine"), especially if the person speaks naturally and without clear enunciation. Errors are compounded by loud environments or simple microphones that may catch surrounding noise. It is often difficult and labor-intensive to produce large enough corpa of relevant phrases to train grammars; not only must we obtain a large number of sample dialogs, but we must also transcribe them all into text. The problem of gathering sufficient data to train a model is more severe in specialized applications, such as our robotic wheelchair.

Even with perfect speech recognition, we would still have the problem of linguistic ambiguities. With a new user or a new environment, we may encounter phrases that we do not expect or are difficult to interpret. For example, different people may describe the same location as a "kiosk," a "booth," or a "desk," and, until the user clarifies what they mean by a certain phrase, we may not know to where they are referring. Another example is that the user may ask the dialog manager about "the elevator," but there may be multiple elevators in the building. Again, the dialog manager must be able to resolve this ambiguity. No matter how much both speech recognition and natural language processing improve, there will always be uncertainty—humans hardly understand each other perfectly!—due to non-ideal recording conditions or novel situations. A good dialog management system must therefore consider the uncertainty, or chance of error, when making decisions on how to interact with the user.

There are several approaches to making decisions in uncertain situations, and in this work we will focus on a planning technique known as a Partially Observable Markov Decision Process (POMDP). Chapter 2 provides a technical overview of POMDPs, but for now we note that POMDPs are probabalistic models in which the agent—in this case, the dialog manager—interacts with the world through a series of actions and receives feedback in the form of noisy observations. In the context of our robotic wheelchair, the actions are primarily queries about the user's objectives and the physical movements of the wheelchair to particular locations. The actions can have unknown effects: for example, if the system asks a user where he wishes to go, it cannot know the answer without hearing his response. Similarly, the observations—processed utterances from the user—may not reflect the user's true intent: the voice recognition system may confuse the similar sounding words. The uncertainties encoded in the POMDP model are uncertainties inherent in the environment; no amount of tuning can make them disappear. A solution to a POMDP tells the agent how it should behave even when these uncertainties are present.

Learning User Models. While some forms of uncertainty are inherent to the system and the environment—such as a noisy microphone—other forms of uncertainty stem from having an incomplete understanding of the user. For example, the system may not know initially that a particular person uses the word "kiosk" to refer to a particular location, and thus it may be confused the

first time it hears the new word. However, if it interacts with the user over and extended period of time (which is likely, if the system is part of a robotic wheelchair), it should ideally adapt to the speech patterns and vocabulary choices of its user. Such adaptation will allow the system to make faster (correct) decisions about the user's true intent based on noisy information. Adaptation can also allow the system to learn the user's preferences. For example, some users may be tolerant of mistakes, while others may prefer that the system ask the user for clarification if it is in doubt about the correct choice of action.

Another form of uncertainty to which the system should be able to adapt is uncertainty about its model of the world. A POMDP model is specified using a large number of parameters, and creating a quantitative user model would be impossible without extensive user tests. As the user interacts with the system, however, it should be able to discover how often certain words are mistaken for other words or the probability of getting a completely spurious output from the voice recognition software. As it learns the true levels of noise in the environment, the dialog manager should be able to make smarter decisions on when to seek additional clarification given a potentially confusing user input.

The ability of the dialog manager to learn dialog strategies robustly online is the core of this work. Just as our dialog model considers uncertainty over the user's true intent, we can also consider uncertainty over possible dialog models. We begin with the basic premise that although specifying the correct model of the user is difficult, specifying a reasonable model of the user is easy. For example, we can guess what words a user will likely use to refer to a particular place or that a user will be extremely frustrated if the robot drives to the wrong location. By taking a Bayesian approach, which allows us to place probability distributions or *priors* over models, we can put higher probability on models that we believe are more likely. These probability distributions can be updated as we gain more information about the true model through user interactions.

Compared to an alternative, where the system starts with absolutely no knowledge of the user, starting with a good initial guess over possible models allows the system to start with a baseline strategy that will make relatively few mistakes as it learns about the user. In Figure 1-1, we plot the performance of an optimal simulated dialog system, a system that learns how to behave with

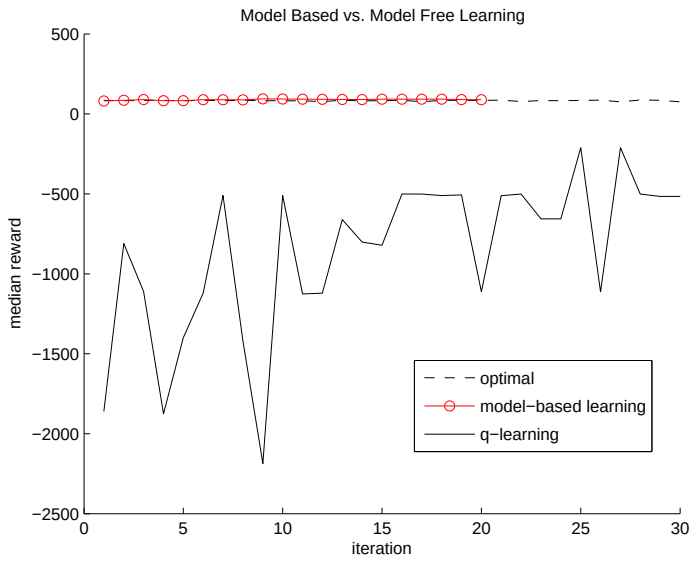

Figure 1-1: A comparison of model-free (Q-learning) and model-based learning approaches for our dialog problem. The model-based approach used here is further described in Chapter 6. The model-based learning approach does start significantly worse than the optimal, but the difference is small compared to how much worse the model-free approach fares.

no prior knowledge (using Q-learning, a standard reinforcement learning technique [38]), and model-learning approach described in Chapter 6. As we will see in later chapters, the model-based approach is actually imperfect; however, here we see that even a naive model initialization starts the learning process at a stage much better than the model-free Q-learning approach. We review various model learning approaches, as well as prior work in Dialog management, in Chapter 2.

Even if we constrain ourselves to Bayesian approaches to learning POMDP models for dialog management, we still must decide how we will express model uncertainty, how we will use interactions to resolve this uncertainty, and how we will act knowing that the model is uncertain. These decisions involve a series of trade-offs: for example, suppose that we assume that the true model is one of three possible user models. With so few models, determining which of the three best fits the user may be simple, but it is likely that the true user model is not a member of this set of models. Conversely, we may allow each of the POMDP parameters to take on all possible (valid) values. Undoubtedly the true model is a member of this class, but with so many possibilites, choosing the right action may be computationally difficult. It may also be difficult to learn effectively when there are a large number of coupled parameters, and the agent may have to make several mistakes

along the way to help it discover the true values of certain parameters.

Our Contributions. In the context of dialog management, we develop a set of techniques that allows us to learn robustly in real-time or near real-time within a rich class of possible user models. Specifically, we develop and demonstrate techniques for quickly approximating policies given a particular user model and for choosing actions when the possible user models disagree. We also introduce "meta-actions," or queries about what the system should have done, as a means to robustly discover the values of certain model parameters. The ability of the agent to reason about its uncertainty—and take advantage of low-risk opportunities to reduce that uncertainty—lead to more robust policy learning.

After review of technical concepts and prior work in Chapter 2, we catalog some insights and algorithms for efficiently solving POMDPs in Chapter 3. This includes sampling approaches that focus solutions to the most relevant aspects of the problem. We also gain significant speed-ups by noting that the symmetries in certain dialog management problems can allow us to exponentially decrease the number of samples required for given quality approximation.

As a starting point into the model-learning problem, Chapter 4 considers only the expected values of the unknown POMDP parameters when planning the dialog policy. We demonstrate a heuristic that allows the dialog manager to intelligently replan its policy given data from recent interactions. While this algorithm is fast and provides a reasonable baseline, we show that this approach suffers because it is unaware of parameter uncertainty and thus can get caught in suboptimal solutions.

In Chapter 5, we fold the unknown parameters into a larger POMDP that jointly considers uncertainty in the user's intent and uncertainty in the true model. First, we focus on learning the user's preferences from a discrete set of options; and even with a small set of options, the resulting POMDP consists of over 300 states. We show that by starting with a suitably conservative prior over models and applying the ideas from Chapter 3, the dialog manager could learn the user's preference model without making many mistakes. Next, we demonstrate similar learning results from a class of continuous models.

The approaches in Chapters 4 and 5 require explicit reward feedback from the user—at least

during a training period, the user has to enter a reward after each action for the planner to learn about the user's preferences. Because it can only learn from direct feedback, the agent is also forced to experience pitfalls in order to learn from them. In Chapter 6, we propose the use of "meta-actions"—queries about what the agent should have done or ought to do—as a means of giving the dialog manager feedback about the user's preferences and speaking style. We show that "meta-actions" allow the system to robustly learn the user model in a rich class of continuous models; in Chapter 7, we support our simulation results with a set of user tests.

We state our conclusions directions for future work in Chapter 8.

Chapter 2

Related Work and Technical Background

Partially Observable Markov Decision Processes (POMDPs) are a model used for planning in stochastic domains where an agent does not have access to direct information about its state. Instead, the agent must make decisions given only noisy or ambiguous observations. If the agent has a probabilistic model of how the world behaves, it can use its belief—a probability distribution over its current state—to reason about what action it should take next. By explicitly tracking this state uncertainty, the POMDP framework allows the agent to trade optimally between information gathering actions (which reduce state uncertainty) and exploitation actions (which gather immediate rewards).

The ability to make robust decisions under the resulting state uncertainty make POMDPs desirable in many areas, including dialog management. Here, an agent must infer the needs of a user (the hidden state) from the noisy and ambiguous utterances that it hears. In this chapter, we begin with a technical introduction to POMDPs in Section 2.1. This introduction presents the standard POMDP model—with no parameter uncertainty—and value function-based solution techniques. [1] Next we describe the basic structure of the dialog POMDP in Section 2.2 before surveying prior approaches to POMDP model learning and dialog management in Section 2.3.

[1] Chapter 4 shows how we incorporate parameter uncertainty into the model

2.1 Technical Background

The POMDP Model. A POMDP consists of the n-tuple $\{S,A,O,T,\Omega,R,\gamma\}$:

- S represents a set of possible states of the world. In the context of robot navigation, states may be physical locations. In the case of dialog management, states may represent the user's true desires. In the POMDP model, the true state of the world is hidden from the agent.
- A represents the set of actions that an agent may take.
- O represents the set of observations that the agent receives.
- The transition function $T(s'|s,a)$ places a probability distribution over possible states s' that the agent may find itself in if it takes action a in state s. For example, if a robot operating in a POMDP world takes the action "move forward" from its current position, it may expect to find itself straight forward or slightly to the left or right, depending on the precision in its movement. Thus, transition function encodes the agent's uncertainty in the effects of its actions.
- The observation function $\Omega(o|s,a)$ places a probability distribution over the possible observations that an agent might receive if it is in state s and takes action a. For example, imprecision in a robot's laser scans may cause it to believe a wall is nearer or farther away than it truly is. Thus, the observation function encodes noise the agent's measurements of the world.
- The reward function $R(s,a)$ states what reward the agent receives for taking action a in state s.
- The discount factor $\gamma \in [0,1]$ allows us to bias the POMDP planner to satisfying goals more quickly. The discount factor weighs how much we value future rewards to current rewards: a discount factor of 0 means that we only value current rewards, while $\gamma = 1$ implies that future rewards are just as valuable as current rewards. Thus, an agent with a small discount factor may overlook policies that require multiple steps to achieve a large reward in the end, but an agent with a large discount factor may dally in achieving the goal.

During each step, the agent first takes an action. This action changes the state of the world as described by the transition function T. The action also produces an observation based on the

observation function Ω. The agent uses the observation to reason about the possible new states of the world and determine what action to take next. Outside of a learning context, where the agent is trying to discover the values of the rewards, the rewards are used only to initially train the agent. The agent does not receive explicit rewards while it is acting in the real-world.

In general, an agent's estimate of the current state, and therefore the best next action to take, will depend on the entire history of actions and observations that the dialog manager has experienced. Since defining a policy in terms of the history can get quite cumbersome, we generally maintain a probability distribution over states, called a *belief*, which is a sufficient statistic for the previous history of actions and observations. No matter how the agent reaches a particular belief, the belief is all that is required to optimally choose the next action. Given a new action and observation, we can update the belief b using Bayes rule:

$$b_n(s) = \eta\Omega(o|s', a) \sum_{s \in S} T(s'|s, a) b_{n-1}(s) \tag{2.1}$$

where η is a normalizing constant. To solve a POMDP, we must specify a policy: a mapping from each belief b to an action a. We define the optimal policy to be one that maximizes the expected discounted reward $E[\sum_n \gamma^n R(s_n, a_n)]$.[2]

Solving POMDPs. There are several approaches to finding the optimal policy of a POMDP. We begin our discussion with value-iteration, the approach we use in our work, and summarize other approaches at the end of this section. Value iteration assigns a utility, or value, $V(b)$ to each belief. Through a series of refinements, value iteration takes some initial function $V_0(b)$ and adjusts it until it until $V(b)$ equals the reward that we expect to get if we begin in belief b and then act optimally. For POMDPs, one can show that the value function is both piece-wise linear and convex [36]. Intuitively, the fact that the value function is convex means that state uncertainty never helps us: we can always expect higher rewards if we know our true state.

Any value function—optimal or not—implicitly encodes a particular policy. Note that given a starting belief b and an action a, there are exactly $|O|$ possible beliefs b_o^a we could transition to in

[2]There are other ways to define optimal policy—for example, maximizing the average reward over a finite number of steps in the future. We use the infinite discounted horizon version of the POMDP largely for convenience.

the next step, one for each observation we may see after taking action a. We can also compute the probability of transitioning to any particular b_o^a, which is equal to the probability of seeing action o after taking action a from belief b:

$$\Omega(o|b,a) = \sum_s \Omega(o|s,a)b(s)$$

where we use $b(s)$ to mean the probability of being in state s according to belief b. Thus, the expected reward from taking action a in belief b is given by the sum of the immediate reward we expect to get and the our (discounted) expected future reward. We give this quantity a special name, $Q(b,a)$:

$$Q(b,a) = R(b,a) + \gamma \sum_{o \in O} \Omega(o|b,a)V(b_a^o). \tag{2.2}$$

where $R(b,a) = \sum_s Rs,a)b(s)$, the immediate reward we expect to get in belief b if we perform action a. Now that we know the value of each action, we can simply choose the action with the maximum expected value. Moreover, the value of taking the optimal action in b must be the value $V(b)$ of being in belief b:

$$V(b) = \max_{a \in A} Q(b,a).$$

Substituting Equation 2.2 into the expression for $V(b)$ we get the Bellman equation for the optimal policy:

$$V(b) = \max_{a \in A} [R(b,a) + \gamma \sum_{o \in O} \Omega(o|b,a)V(b_a^o)]$$

We will solve the Bellman iteratively using dynamic programming. Suppose that we only had one action left to do; clearly, we would choose the action that maximized the expected immediate reward. Let R_a represent the vector of rewards for each state if we take action a. Then optimal final action is $\arg\max_a R(\cdot,a) \cdot b$. Note that this function is piecewise linear in the belief b. Now, suppose that at some later stage, the value function is still piecewise linear. In particular, we will represent the value function at the nth iteration a as collection of alpha vectors $V_n = \{\alpha_1, \alpha_2, ...\alpha_k\}$. These vectors represent hyper-planes in an $|S|$-dimensional space; the value of a belief is given by

$V_n(b) = max_\alpha b \cdot \alpha_i$. If we rewrite the Bellman equation in vector form, we find:

$$V_{n+1}(b) = \max_{a \in A}[R_a \cdot b + \gamma \sum_{o \in O}(\sum_s T_{s',a}\Omega(o|s',a)). \cdot \alpha_a^o \cdot b]$$

where α_a^o is whatever alpha vector in V_n that maximized the value of b_a^o. Note that we can factor the belief b from the expression to get a linear expression in b:

$$V_{n+1}(b) = \max_{a \in A}[[R_a + \gamma \sum_{o \in O}(\sum_s T_{s',a}\Omega(o|s',a)). \cdot \alpha_a^o] \cdot]b \tag{2.3}$$

Thus, by induction, the value function is piecewise linear. We also have an expression for an alpha vector in the revised solution. We note that the alpha vector also has an associated action: if the a is the action that is the $\arg\max$ of the expression above, then it is the optimal action associated with that alpha vector.

Each iteration, or *backup*, represents planning one more step into the past and brings the value function closer to its optimal value[10]. Unfortunately, when we do a backup, we must apply Equation 2.3 to every belief in an infinite belief space. As a result, the number of alpha vectors grow exponentially with the number of iterations. For the infinite horizon, the exact solution may consist of an infinite number of vectors! Even with a finite horizon, however, we may quickly run into trouble: if there are n alpha vectors in one iteration, the next iteration may have up to $|A|n^{|O|}$ alpha vectors. State of the art algorithms for solving POMDPs exactly limit the size of the solution with linear programming and other techniques to prune redundant alpha vectors [1]. However, even the best exact approaches may take hours to solve on fairly small problems; they typically do not scale over tens of states.

Since the exact solution to equation 2.3 using an iterative backup approach is exponentially expensive, we approximate the true backup operation by backing up at only a small set of beliefs[26],[27]. The approximation approach and the choice of beliefs determine the quality of the solution and thus the performance of the dialog manager. By approximating both the value at a particular belief and its derivative, several approaches, including "Point-Based Value Iteration" (PBVI) [26] and "Perseus" [37], are able to produce approximations that generalizes well to other parts of the belief

space. Note that since the solution to the value function is piecewise linear, the derivative of the value function at a particular point is exactly the alpha vector for that point. Therefore, one way to think about the point-based approximations is that instead of collecting all the alpha vectors, we only look for alpha vectors for certain beliefs that we think will represent the entire space well.

A point-based backup consists of the following steps:

1. First, we project the alpha vectors forward for each possible action, observation pair:

$$\Gamma^{a} = \{\alpha | \alpha(s) = R(s,a)\} \tag{2.4}$$

$$\Gamma^{a,o} = \{\alpha | \alpha(s) = \gamma \sum_{s'} T(s'|s,a)\Omega(o|s',a)\alpha'(s)\}, \forall \alpha' \in V_{n-1} \tag{2.5}$$

   The Γ^{a} and $\Gamma^{a,o}$ sets represent the $R(\cdot, a)$ and $\gamma \sum_{o \in O}(\sum_{s} T(s'|\cdot, a)\Omega(o|s', a)) . \cdot \alpha_{a}^{o}(\cdot)$ parts of Equation 2.3, respectively. So far, we have not limited our solution based on our limited belief set.

2. Next, we find the best combination of the the gamma sets for each belief. These are our Q-vectors:

$$\Gamma^{a,b} = \{\alpha | \alpha = \Gamma^{a} + \sum_{o} \arg\max_{\alpha \in \Gamma^{a,o}} (\alpha \cdot b)\} \tag{2.6}$$

   At this point, we have simplified our computations because instead of computing the Q-vector for all of the beliefs, we have found the Q-vector for our limited set of beliefs.

3. Finally, we take the dominating Q-vectors be part of our new set of alpha vectors:

$$V_{n} = \{\alpha | \alpha = \arg\max_{\alpha \in \Gamma^{a,b}} (\alpha \cdot b)\}, \forall bin B \tag{2.7}$$

   This step is exactly what we would have done in a standard update.

Choosing an appropriate belief set is an active area of research. PBVI recommends starting with some belief b_0 (such as being in a 'dialog-start' state). Then for each action a, we sample a user response o from the observation distribution and compute the updated belief state b_{o}^{a} (simulating the effect of one exchange between the user and the dialog manager). We add the farthest

new beliefs to our set and repeat the process until we accumulate the desired number of beliefs. Since the beliefs represent confusions over the user's intent, picking beliefs reachable from the starting belief focuses our computation in situations the dialog manager is likely to experience. PBVI backs up all beliefs in each iteration. Other belief sampling approaches use more complex heuristics to iteratively add points that are most likely to cause a change in the current value function. Regardless of how the belief set is chosen, one can show that given sufficient iterations, the final error in the value function will be bounded by the largest distance between some reachable belief and a member of the belief set.

Perseus takes a somewhat opposite view: instead of trying to determine which belief points will be the most important, and backing those up, it first creates a very large set of reachable beliefs. However, instead of trying to backup every one of these beliefs, Perseus only backs up enough beliefs in each iteration to uniformly improve the value function. Although more updates are needed to each a full backup, each update can be very fast. Perseus also avoids the dilemma of how to choose an small support belief set. The randomized backups still provide monotonic improvement in the value function, although it is more difficult to determine how long one must run the algorithm to achieve a particular level of performance.

Finally, we note that many optimizations have been made to improve upon these (now standard) approaches to value iteration. In particular, "Heuristic Search Value Iteration" [35] speeds up convergence to the value function by maintaining both a lower bound (using alpha vectors) and an upper bound (using linear programming). It chooses to backup the value function in regions that will minimize the maximum difference between the upper and lower bounds on the value function. We chose not to use HSVI primarily for reasons related to the ease of implementation, but if we continue to value iteration in our future work, HSVI does provide an alternative to PBVI and Perseus that comes with stronger performance guarantees.

Point-based approximations to value functions are an attractive way to solve POMDPs because they offer a simple dynamic programming approach that can be efficiently implemented using matrix operations. However, there do exist other approaches to solving POMDPs. The main other approach is policy iteration. Instead of representing a policy implicitly through a value function,

some policy based approaches typically represent the policy more explicitly using a finite state controller (FSC). FSCs are graphs which consist of a set of action-nodes. Observations transition the agent from one action node to the next. The advantage of FSCs is that they can often represent policies compactly and converge to near-optimal policies in fewer steps than using value iteration [17], [11]. FSCs can also be used to find patterns and hierarchies in the POMDP structure that further reduce the size of the policy [2]. However, solving FSCs for a representation of a given size can be quite difficult, and in general FSC optimizations require more complex computations (such as semi-definite programming).

Other policy iteration methods [23], [16] use a tree-based representation. Given a current starting belief b_0, they first consider the set B_1 of the $|A||O|$ beliefs they may encounter after taking one action and receiving one observation. Next they consider all the beliefs B_2 that they may encounter if they take an additional step from each the beliefs in the set B_1. In this way, they construct a tree of beliefs that are reachable in a fixed number of steps. Each node in the tree represents a belief, and we can compute the expected immediate reward for each belief. Given these values and the probability of encountering each belief from the starting point b_0, we can compute the value of b_0 (the discount factor allows us to ignore beliefs beyond a certain depth.) The benefit of tree based approaches is that the size of the state space no longer matters; we only plan ahead from our current point. The primary drawback is that the trees may have to be fairly deep to get a reasonable approximation of the value function, and thus a large amount of computation may be required at each step.

2.2 The Dialog Management POMDP

There are many ways of expressing POMDPs for dialog management, and here we describe one simple model that we will use throughout this work. Since we are interested in an application for a robotic wheelchair, our states are the set (hidden) user intentions—in particular, the locations where the user may wish the robotic wheelchair to go. Actions include queries to the user and physical movement. In our basic model, the agent can choose from two kinds of queries: it can choose to ask a general question (such as, “Where would you like to go?”) or confirm a specific goal (such

as, “Did you want to go to the elevator?”). Based on the most recent action and the user’s (hidden) state, the agent has a model of what observations it may receive. Our dialog manager searches the output of the voice recognition software [9] for predetermined keywords to use as its observations. We will assume that the state, action and observation sets are all discrete and finite (although work has extended dialog managers to large state spaces [39] and continuous observation spaces [40]).

The transition function $T(s'|s,a)$ states what the user is likely to want next, given the state they were just in and the action the system just took. In all of our dialog models (see Figure 2-1), we assume that the user is unlikely to change their intent mid-dialog. Thus, the most likely user state sequence is (1) initiating dialog, (2) seeking a goal location, and (3) completed dialog. The transition probabilities between goal states is fairly small. The observation function $\Omega(o|s,a)$ encodes both the set of words that a user is likely to use to describe a particular goal state and the speech recognition errors that are likely to occur. Finally, the reward $R(s,a)$ gives us a way to specify what the “right” actions are in different states and how much the user is willing to tolerate clarification questions before becoming frustrated.[3]

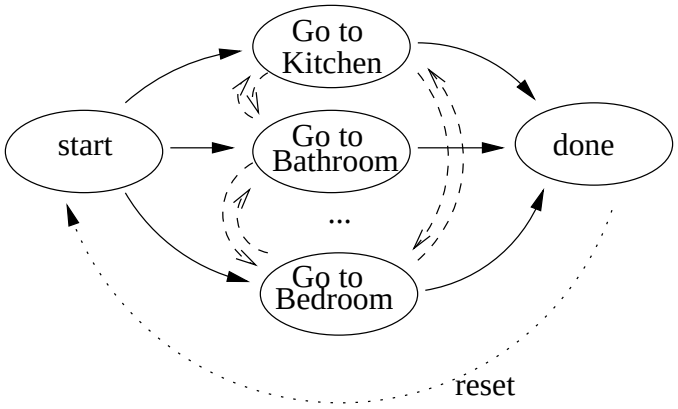

Figure 2-1: A toy example of a dialog POMDP. Solid lines represent more likely transitions; we assume that the user is unlikely to change their intent before their original request is fulfilled (dashed lines). The system automatically resets once we enter the ‘done’ state.

[3]Since repeated questions can also be frustrating, we did briefly explore extending the basic state model to include a counter for how often a query has been asked and impose larger penalties for repeated questions. However, we found that effects of the frustration model to be relatively small. Thus, we opted to use the more computationally tractable basic model in our tests.

2.3 Related Work

In this section, we review literature from fields most related to our work. First, we survey work in learning POMDPs with parameter uncertainty. Next, we review prior applications of POMDPs for dialog management. We conclude our review with examples of other approaches to dialog management.

Learning in POMDPs with parameter uncertainty Several works have considered the problem of planning with uncertain parameters in the POMDP or MDP framework. Closest to this work are the Bayesian approaches of [15] and [30]. Both of these approaches place priors over the model parameters and update them as observations are received.

The Medusa algorithm[15] extends the observations of [4] (who suggests sampling MDPs and updating their value functions as new data received). Taking a particle-filter approach to the problem, Jaulmes et. al. sample a set of POMDPs from an initial distribution over models. As the prior distribution is updated, each POMDP is reweighted according to its likelihood; occasionally low-weight POMDPs are discarded and new models are sampled from the current model distribution. Actions are chosen stochastically: each POMDP applies its weight toward the action that its policy recommends. The advantage of Medusa's approach is that the small POMDPs can be solved quickly. In fact, with so many POMDPs, the overall policy will often be correct even if each POMDP's solution is not fully converged. The disadvantage to the Medusa is that the action selection does not consider how an action may change the model prior; in this sense Medusa is still blind to the uncertainty in the parameters.

The Beetle algorithm[30] takes a decision-theoretic approach to solving MDPs with uncertain parameters. Using recent advances in continuous-POMDP solution techniques[12], Poupart et. al. treat the unknown MDP parameters as hidden state in a larger POMDP. Since the state is always known, the transition statistics for the MDP can be readily updated; these updates change the agent's belief over possible MDP models. While it may be possible to extend such an approach to POMDPs, planning over a high-dimensional, doubly-continuous state space—that is, continuous in both the POMDP parameters and in the belief state—would be quite computationally difficult,

and likely to yield poor solutions without additional work in POMDP value function approximation techniques.

Others take non-Bayesian approaches to solving MDPs with uncertain transition matrices ([24],[42]). Aimed toward industrial applications, where each machine may have different parameters within a given tolerance, these approaches consider the space in which the MDP may reside and find the policy with the best worst-case performance over the set. Learning is not incorporated into the model, but in some cases, the user can trade-off between robustness (worst-case performance) and maximizing performance with respect to a nominal model[42]. While not directly related to our approach (and currently limited to MDPs) these approaches do provide a means of ensuring robust behavior for applications where the user may be willing to receive lower average-case performance in order to ensure that large errors occur with very low probability.

Finally, we note that there do exist model-free approaches for learning in partially observable environments. Early work applied various heuristics [20] to Q-learning [38], originally a method for learning in fully-observable environments (that is, cases where the agent can determine its state exactly through observations). These heuristics included using an initial model to seed the value function and then constraining the value function to include a fixed number of alpha vectors. More recent work has shown that it is possible to learn the optimal policy both without a model and without the ability for the agent to reset itself and try again [7]. Lastly, other works discard the notion of an underlying POMDP model entirely and create structures that learn and plan based only on previous action-observation histories and expected future action-observation histories [14]. While these works improve upon the Q-learning plot in Figure 1-1, we note that all of them require a large amount of data to produce reasonable results even on small problems, and, without a prior notion of possible pitfalls, they are likely to make many mistakes during the learning process.

POMDPs for Dialog Management. The ability to manage the information gathering trade-off have made POMDP-based planners particularly useful in dialog management, including in health-care domains ([33], [13],[8]). These range from a nursebot robot, designed to interact with the elderly in nursing homes[33] to a vision-based system that aids Alzheimer's patients with basic tasks such as hand-washing [13]. Others [8] note that in many cases, the aiding agent's policy can

be simplified if one assumes that just waiting—that is, doing nothing and simply observing the user—will resolve the system's current uncertainties.

Indeed, much POMDP dialog-management research has focused on developing factored models and other specialized structures to improve performance and algorithmic complexity ([40], [28], [39]). To improve performance, [40] incorporate an explicit confidence output from the voice recognition system as an additional measurement. Thus, they can reason more effectively about the quality of the observations they receive. In situations where only certain actions are relevant to certain states, the POMDP can be factored into hierarchies that reduce the overall amount of computation required [28]. These approaches typically assume a reasonably accurate user model. In domains where large amounts of data are available—for example, automated telephone operators—the user model may be relatively easy to obtain. For human-robot interaction, however, collecting sufficient user data to learn a statistically accurate model is usually expensive: trials take a lot of time from human volunteers.

While none of these dialog-management works discuss learning user models, insights to decreasing computational load will be crucial to our work (especially since we propose to incorporate the unknown parameters into a larger POMDP). The Summary POMDP algorithm [39], designed for slot-filling dialogs for automated telephone operators, approximates a policy by assuming that the only important feature of the belief is the probability of the most likely state. This probability can be used to determine whether the system should make a general query, confirm (the most likely state), or submit (the most likely state), and in practice this allows Summary POMDP to handle large dialogs. In follow on work, the authors describe how to further streamline the solution procedure to handle even larger dialogs [41]. We will make similar observations about the symmetries present on our dialog POMDPs in Chapter 3.

Other Approaches to Dialog Management. While POMDP-based systems are the focus of our work, there are many other approaches to dialog management. Typically, these involve a set of rules that the dialog manager will follow given particular outputs from a voice recognition system (or equivalently, a finite-state machine). For example, the Mercury system [34] builds a network in which a flight reservation system keeps track of what information has already been provided and

for what the user needs to be prompted. In the context of flight reservations, it must also guide the user to choose a valid (that is, available) flight. Some systems use a rule-based model, but add a predictive layer that senses when the conversation is likely to have degraded (based on pauses from the user, low confidence scores from the voice recognition system, etc.) [25]. If the dialog degrades below a certain threshold, then the system will call upon a human operator for assistance.

Just because a system is based on rules does not imply that it cannot adapt to its users. The TOOT system [19] classifies training dialogs as successful or unsuccessful using various thresholds, and then adapts its rules, which are based on various features of the dialog, to improve performance. Another learning approach first trains an AdaBoost classifier based on a large set of training data [3]. Thus, given a variety of inputs, the system provides the response that the user most likely needed (based on the output of the classifier). These systems may need complex procedures to handle uncertain inputs robustly, or may ignore uncertainty all together. Ignoring uncertainty may be acceptable in automated telephone dialog systems, where repeating a request is easy and the consequences of servicing a request incorrectly are relatively small.

Other systems come closer to the POMDP model, for example, the NJFun system [18] is designed to learn over time. NJFun learns parameters from scratch during a training period. The actions are specifically designed so that reasonable dialogs may be maintained during this period. The execution of this model is facilitated by an MDP model that includes ‘buckets’ for different levels of uncertainty (effectively a highly-discretized POMDP). However, like all of the approaches above, the NJFun system is not aware of the uncertainty in its parameter and thus cannot take actions to reduce that uncertainty. It also only learns during a specified training period.

Chapter 3

Special POMDP Approximations

Despite the advances in POMDP approximation techniques described in Chapter 2, solving a POMDP—even when all the parameters are known—is still a highly non-trivial task. In this chapter, we describe a collection of insights to improve the speed and accuracy of our solutions. Section 3.1 contains suggestions on heuristics for choosing belief samples. Section 3.2 shows how, if the POMDP contains certain symmetries, we can solve it with exponentially fewer belief points by mirroring sections of the belief space.

3.1 Sampling Heuristics

As we saw at the end of Chapter 2, state-of-the-art POMDP solution techniques approximate the entire value function by finding its gradient at a limited set of belief points. Different algorithms have different ways of choosing these points: PBVI [26] computes all one-step trajectories from each point in the belief set and adds the belief that was farthest away from the set. Perseus [37] creates a very large set of all reachable points. In this section, we discuss what sampling techniques worked best for our problem. In particular, solving a dynamic POMDP—that is, one where the parameters are changing as we learn more about the environment—adds several considerations when choosing belief points.

3.1.1 Sampling for the Static Problem

The dialog problem is interesting because unlike robot exploration, we are not trying to rapidly expand to certain corners of the belief space: there are, in fact, a very limited number of states and we know exactly how to get to them. The dialog management problem, as we have formulated it in Chapter 2 is trivial in the absence of noise; no complex paths must be planned from the start to the goal. A rapid expansion to far away states is less critical than collecting beliefs that express confusions we are likely to see due to noisy utterance observations.

One pitfall that we wish to avoid is making sure that the POMDP solution is not too afraid to act. Since we are initially unsure of the user's intent and the observations are noisy, we will never be a hundred percent sure about the user's intent. If we do not sample deeply enough, and the severity of making a mistake is large enough, we may decide never to risk making a mistake. (For example, if our most certain belief is 99% certain of the goal state, a penalty of -1000 for acting will mean that the optimal action in that belief is still to confirm that location.) We can avoid this problem by including all the corners of the belief simplex—that is, all beliefs where the state is certain—in our initial belief sample. The effect of these points is to make our policy less conservative (necessarily, since they add hyper-planes in which we choose to act rather than gain information).

We generally want our POMDP to eventually commit to some physical movement, but in Chapter 5, we will see a situation where this effect is not necessarily desirable. There, we have a large POMDP with 336 states. Solving the POMDP takes a long time because we need many more beliefs to span the space. With 336 states, we can no longer afford to sample belief points densely enough to meet any kind of useful performance criteria, and we must consider carefully how we should pick our belief points. For the simple case above (used in Chapter 4), we seeded our initial belief set with corner points and common confusions that we wanted our POMDP to handle. If we follow the same approach here, we find that the resulting policy is too aggressive. In the simple POMDP, the seven corner points made up only a small fraction (1.6%) of the 500 total beliefs. However, even if we use 1500 samples now (a limit we chose for making the simulations run in a reasonable amount of time), including all 336 corner beliefs means that corner beliefs make up

more than 20% of the total beliefs. The remaining beliefs could be sparsely scattered throughout the simplex, and therefore, the most influential sampled belief to a particular point is often a corner belief. We will resolve this issue in the next section using dynamic sampling techniques.

3.1.2 Smart Sampling for the Dynamic Problem

Given that our knowledge about the POMDP parameters are changing, it is only natural that the changes to the model will impact what beliefs are needed to make a good approximation: beliefs that we had never encountered before may suddenly become common under a new set of observation parameters. For example, with symmetric and unimodal observations, the belief is usually peaked around a particular state. If the true observation model is bimodal, then we will find our solution quality is poor not due to the number of backups, but because bimodal beliefs were underrepresented in our original sample set. There are two ways we can attack the problem of what belief points we should use in our sample: either we can find a sample that will do well in all situations, or we can adapt our sample as we get more information about the unknown parameters.

For small problems, we work around this issue by seeding the initial belief set with not just the starting belief, but a set of beliefs that include situations—such as bi-modal and tri-modal beliefs—that we would like our dialog manager to be able to handle well, regardless of the true model parameters are. When the problem is small, hand-picking key beliefs efficiently improves the quality of the solution.

For large problems, resampling beliefs is key. Given a state space with 336 states, we really cannot hope to do it justice with only 1500 belief samples, even if we try to pick the points wisely—we used 500 samples for the basic problem before. We need a robust solutions that can still be solved quickly. In our particular case, the state space explosion occurs because we will be considering joint uncertainty in the model parameters and the user state. As we learn about the true model, our belief converges to smaller parts of the belief simplex. Therefore, we really do not need to know how to behave in every possible parameter set: at the beginning, we only need to know how to behave in our start state, and as we learn more about the parameters, we can refine our policy for the relevant part of the state space. By resampling as we go, even though our POMDP

never changes, our policy still 'adapts' in the sense that we refine the policy around parts of the belief space that are actually relevant. Table 3.1.2 summarizes our approach.

Table 3.1: Algorithm for dynamic resampling for iterative solutions to large POMDPs. While generally applicable, this technique works particularly well if the reachable belief space becomes smaller over time.

DYNAMIC RESAMPLING
• Sample a fixed number of belief points from the starting belief (using PBVI or any other sampling approach). • Solve the POMDP. • Loop: – Interact with the user. – Resample a new belief set given new starting belief. – Perform additional backups on the POMDP solution to refine it around the new region of interest.

3.2 Fast Solutions to Symmetric POMDPs

One reasonable starting point for the dialog POMDP is to say that observations and rewards are symmetric and unimodal. For rewards, this means that the reward for going to the right location is the same in every state, as is the penalty for doing an incorrect action. There are certainly situations where this is not true—for example, the user may be less annoyed if the wheelchair seems to mistake similar sounding words—but it is a reasonable starting point.

Likewise, symmetric observations imply that the probability of hearing noise is the same in all states. Unimodality means that the distributions have one peak around the most likely observation and all other observations are equally unlikely. Again, observation symmetry is definitely not true in the real world, where similar sounding words are much more likely to be confused that dissimilar words. However, it is a reasonable approximation often used in practice [39], [33]. We will not try to further justify the symmetric, unimodal observation model, but we show that within such a model, the POMDP can be much simpler to solve.

Finally, we note that our dialog POMDP has a very special action structure. There are three kinds of actions, some of which take a state as a predicate: ask (a general question), confirm (a specific state), and act (go to a specific location). Thus, a symmetry also exist in the actions.

More formally, we consider the class of POMDPs with the following properties. Let the true state of the user be called the target state s_t. Let $o(s, a)$ the most likely observation in state s after taking action a. Let $a(s)$ denote taking action a with predicate state s (for example, a might be "confirm" and s might be "cafe"). For each action in the symmetric POMDP, the rewards, observations, and transitions are given by:

- $R(s, a(s'))$ = R_a^+ if s = s' and R_a^- otherwise.
- $\Omega(o|s, a)$ = p_a^+ if o = $o(s, a)$ and p_a^- otherwise.
- $T(s''|s, a(s'))$ = q_a^+ if s = s'' and q_a^- otherwise (an information gathering action probably does not change the state) *or* q^* if s = s' (a correct "act" action may reset the probabilties over target states).

Note that both rewards and observation probabilities do not depend on the state.

If we ignore the "start" and "done" states in our dialog model, then the remaining target states follow the properties described above. For example, consider the following two beliefs over three possible target states: $b_1 = (0.5, 0.3, 0.2)$ and $b_2 = (0.3, 0.5, 0.2)$. If the correct action in the first belief is to confirm the first state, then the correct action in the second belief must be also to confirm (in this case, the second state). The ordering of the belief values do not matter when choosing the action type, only the values themselves. The predicate for the action type is always the most likely state. It should therefore be clear that in these special POMDPs, the identity of the state does not affect the correct action type (although it may affect its predicate).

This observation has profound implications for solving the POMDP. What we have just said is that given a belief $p_1...p_n$, any permutation of that belief has the same action type. Note that permutations are exponential in size; thus, knowing the correct action for one belief tells us about how we should behave in exponentially many other beliefs (except for the predicate, which is easy

to attach later). Without loss of generality, we will show how we can solve for the entire value function using only beliefs that whose probabilities are sorted in non-decreasing order.

Before we describe our solution procedure, we note that we cannot simply take a small set of sorted beliefs, compute value-backups on them, and expect to get the correct value function. Consider a simple two state situation, where the actions are "Commit to State One," "Commit to State Two," and "Query for Correct State." Suppose that we are in the belief $b = (0.5, 0.5)$. From the problem symmetry, it should be clear that the values of beliefs $b^{o_1}_{ask}$ and $b^{o_2}_{ask}$ should be equal. However, suppose that we only include beliefs that place greater probability on the second state. The action "Commit to State One" will never be chosen in this setting, and without knowing that positive rewards are available in the first state, the value function will place lower value on $b^{o_1}_{ask}$ than $b^{o_2}_{ask}$. More generally, $V(b) < V_{opt}(b)$ since for any belief that prefers state two, there is some probability that we may transition to a 'bad belief' that favors state one.

We now describe our procedure for computing the optimal value function using only the sorted set of beliefs B_s. Suppose first that if some alpha vector α is part of the solution to the value function at iteration n, then all permutations $\{\pi(\alpha)\}$ are also part of the solution to the value function at the n^{th} iteration (we will prove this statement shortly). Since all of the beliefs in our sample have non-decreasing values, the permutation $\pi^*(\alpha)$ that sorts α in non-decreasing order will maximize the dot product $\max_{\pi(\alpha)} \alpha \cdot b$. Let the value function V_n be the union of all permutations of a set of alpha-vectors $\alpha_1..\alpha_k$:$V_n = \cup_{i=1}^{k}\{\pi(\alpha_i)\}$. Then, to find the alpha-vector that maximizes the dot product with some sorted belief b, it is sufficient to consider the (exponentially smaller) set of alpha vectors $V_n = \cup_{i=1}^{k}\pi^*(\alpha_i)$. This observation immediately suggests the following procedure for computing V_{n+1} for our sorted set of beliefs:

1. Propagate each alpha-vector in V_n forward to create the $\Gamma^{a,o}$ sets. Note the number of alpha-vectors in V_n for the sample set B_s is at most $|B_s|$.

2. Within each $\Gamma^{a,o}$ sets, sort the g vectors in non-decreasing order. Note that we have note increased the size of the sets. (We will show that if all permutations of the alpha-vectors are present in the solution V_n, then all permutations of the g vectors are present in the $\Gamma^{a,o}$ sets; recall that only the permutation of the g vector that has non-decreasing values may maximize

the dot product $\Gamma \cdot b$.)

3. Choose the combinations of the $\Gamma^{a,o}$ to construct the new alpha-vectors for V_{n+1}.

Note that once we can compute the correct value $V(b)$ for $b \in B_s$, we can compute the value of any other belief b' by first sorting it and then evaluating it through our value function.

It remains to be shown that for any alpha-vector in the solution V_n, all permutations of that alpha-vector are also part of the solution V_n and all permutations of the g vectors are part of the sets $\Gamma^{a,o}$. We provide the following sketch of the argument. In the first iteration, let us set the alpha vectors to be reward vectors R_a. By the problem symmetry, if the reward vector $R_a = (r_a^+, r_a^-, r_a^-)$ for some action, then the reward vectors $R'_a = (r_{a'}^-, r_{a'}^+, r_{a'}^-)$ and $R_{a''} = (r_{a''}^-, r_{a''}^+, r_{a''}^-)$ also exist in the set, that is, if a vector R is present, all permutation of R are also present.

For the inductive step of the argument, suppose that at some iteration n, we have a collection of alpha vectors $\alpha_1..\alpha_k$ and a corresponding solution $V_n = \cup_{i=1}^{k}\{\pi(\alpha_i)\}$. To compute the $\Gamma^{a,o}$ sets:

$$\Gamma^{a,o} = \{g | g(s) = \gamma \sum_{s'} T(s'|s,a)\Omega(o|s',a)\alpha'(s)\}, \forall \alpha' \in V_{n-1} \tag{3.1}$$

Suppose that some α_i was used to compute a vector $g_i^{a,o} \in \Gamma^{a,o}$. By the symmetry in the problem note that $g_i^{a',o}$ is a permutation of $g_i^{a,o}$ if a is of the same type as a'. Also by problem symmetry, note that a permutation of α_i will produce a permuation of $g_i^{a,o}$. In this way, we can argue that all perumtations of $g_i^{a,o}$ are present among the $\Gamma^{a,o}$ sets.

The remaining steps are simple: consider forming the sets $\Gamma^{a,b}$:

$$\Gamma^{a,b} = \{g' | g' = \Gamma^a + \sum_o \arg\max_{g \in \Gamma^{a,o}} (g \cdot b)\} \tag{3.2}$$

Note that although we are only considering sorted beliefs, all perumations of those beliefs are valid beliefs. Thus, suppose that $b \in B_s$ picks out a vector g as the argmax. Then, a permutation of b will pick out a permutation of g as the argmax. Thus, if g is the optimal vector for some belief, then all permutation of g will also be part of the optimal set. An identical argument holds when we choose the best $\Gamma^{a,b}$ vectors to be included in the value function V_{n+1}.

Thus, we can conclude that all permutations of any particular alpha vector exist in the set. Therefore, we can plan using only sorted beliefs and sorted alpha vectors. While this is only useful if the problem contains certain symmetries, in practice, many dialog problems can be modeled this way (even if they are not exactly symmetric). Indeed, this framework should apply to many preference elicitation problems, and other problems where the primary uncertainty is to discover what (static) state we are in, and where we have actions that help us to sense generally, probe for a particular state specifically, or commit to a particular state. For example, problems of trying to detect the presence of a particular trace element or the identity of a material may also fall into this category.

As a validation of the performance of this approach, Figure 3-1 shows the performance of using the permuted approach on the dialog problem with 48 different choices of parameters. In the PBVI case, 500 belief points were selected and backed up 25 times. Note that the solution does poorly, although adding additional backups (up to 200) improves the quality of the policy significantly. The permuted case only had 24 belief points and was also backed up only 25 times. Even with such little computation, it performs much better than the 500 point sample set and almost as well as the set with 8 times as many backups.

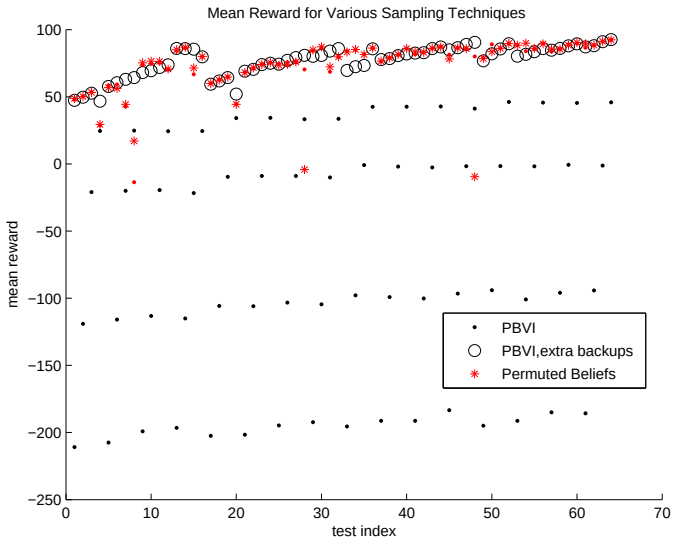

Figure 3-1: Performance of Permuted Approach compared to standard PBVI sampling. The permuted approach had only 24 sample points, compared to 500 for the PBVI set, and achieved good performance with 8 times as little backups.

Chapter 4

Maximizing Expected Performance

In Chapters 2 and 3, we argued that POMDPs provided a good framework for decision-making in stochastic environments and described how, given a POMDP, an agent should solve that POMDP to determine a policy of optimal behavior. We now turn to the main focus of this work: how should an agent behave if it does not know all of the parameters in the POMDP? Taking a Bayesian approach, we begin by placing priors over the unknown parameters. These priors allow us to encode knowledge and intuition we may have about the problem. For example, while we may not know the exact reward parameters of the true POMDP, we can guess that an incorrect movement will have a higher penalty than a clarification question. Placing priors over the parameters induces a distribution over possible models. Section 4.1 describes more precisely the prior distributions that we place over the parameters.

In this chapter,[1] we begin an exploration of two issues surrounding planning when we have a distribution over models instead of one known model. The first issue is one of action selection. In Section 4.2.1, we show that, in the absence of learning, the agent should plan its policy using the expected values of the uncertain parameters if it wishes to maximize its expected reward. The second issue relates to improving the agent's knowledge of the model: as it learns more about the model parameters, its distribution over possible models should peak around the true POMDP model. In Section 4.2.2, we describe more precisely how we can update our belief about the model

[1] Work in this chapter was previously presented in [5].

parameters given data from an interaction. We conclude our approach in Section 4.2.3, where we present efficient update heuristics to adjust policy that we found in Section 4.2.1 after another user interaction.

We note that the idea of having a distribution over possible models is very similar to having belief over possible states. Indeed, we will see these two concepts merge in Chapter 5. For the present, however, we will consider state uncertainty and model uncertainty separately—planning in an uncertain state space is already challenging, and planning in a joint uncertain state-model space poses additional computational challenges. We also note that since our planning step in Section 4.2.1 uses only the expected values of the uncertain parameters, our agent's policy is unaware of any parameter uncertainty. In Section 4.3, we show that even this basic form of learning and policy refinement produces a more adaptable dialog manager.

4.1 Model Definition

We use the simple POMDP dialog model described in Section 2.2. In particular, our model consists of five goal locations on the first floor of our building: the Gates Tower, the Dreyfoos Tower, the parking lot, the information desk, and the Forbes cafe. Taking a Bayesian approach, we place priors over the reward, observation, and transition parameters. We describe the form of the priors in this section, and in Section 4.2, we describe how we update these priors as new data arrives (and how we adapt our dialog manager's policy to the changing distributions). We assume that the discount factor was fixed and known.

Rewards. We place a Gaussian distribution over each reward parameter $R(s, a)$. The Gaussian is initialized with a mean, a variance, and "pre-observation count." The pre-observation count measures our confidence in our estimate of the mean: it corresponds to how many "faux-observations" from which the data was obtained. For example, if we are quite confident that the penalty for asking a question r_{ask} is -1, we may set the pre-observation count to 100, that is, we are as sure that $r_{ask} = -1$ as if we had seen the user enter "$r_{ask} = -1$" a hundred times. Conversely, if we are unsure of the value of r_{ask}, we may set its pre-observation count to 10 (or even 1 if the value

was really just a guess).

Since our current approach uses only the expected values of the parameters, the rewards may be either stochastic or deterministic. For our purposes, however, the initial choice of the reward variance does not reflect the inherent stochasticity of the system (that is, an inconsistent user). Rather, the variance reflects our certainty our current expected value of the reward. A high variance suggests that that we are not very sure about our reward value, whereas a low variance suggests that we are confident about our reward value. While partly redundant with the pre-observation count, we note that observation counts alone are not a good measure of parameter certainty. A new measurement will always increase the observation count. However, our updated variance will differ based on how closely the new measurement matches the previous measurements. In particular, if a new measurement does not match new values, we would like our system to become less certain about the true value.

Observations and Transitions. We capture the uncertainty in the transition and observation parameters using Dirichlet distributions. Recall that for discrete states and observations, T and Ω are multinomial distributions $T(s'|s,a)$ and $\Omega(o|s,a)$. Figure 4-1 shows an example of a simplex for a discrete random variable that can take on three different values. Every distribution over those three variables is a point on this simplex: for example, the point (0,.5,.5) represents a distribution where the first value never occurs and the second two values are equally likely. The Dirichlet distribution is a natural choice because it places a probability measure over all valid multinomial distributions. As the conjugate prior for the multinomial distribution, we will see that the Dirichlet distribution is also easy to update given new observation data.

Given a set of parameters $\theta_1 ... \theta_m$, the likelihood of the discrete probability distribution $p_1 ... p_m$ is given by

$$P(\underline{p}; \underline{\theta}) = \eta(\underline{\theta}) \prod_i^m p_i^{\theta_i - 1} \delta(1 - \sum_i^m p_i),$$

where η, the normalizing constant, is the multinomial beta function. The Dirac delta function $\delta(\cdot)$ ensures that the probability of $\underline{p}$ is zero if $\underline{p}$ does not represent a valid probability distribution, that

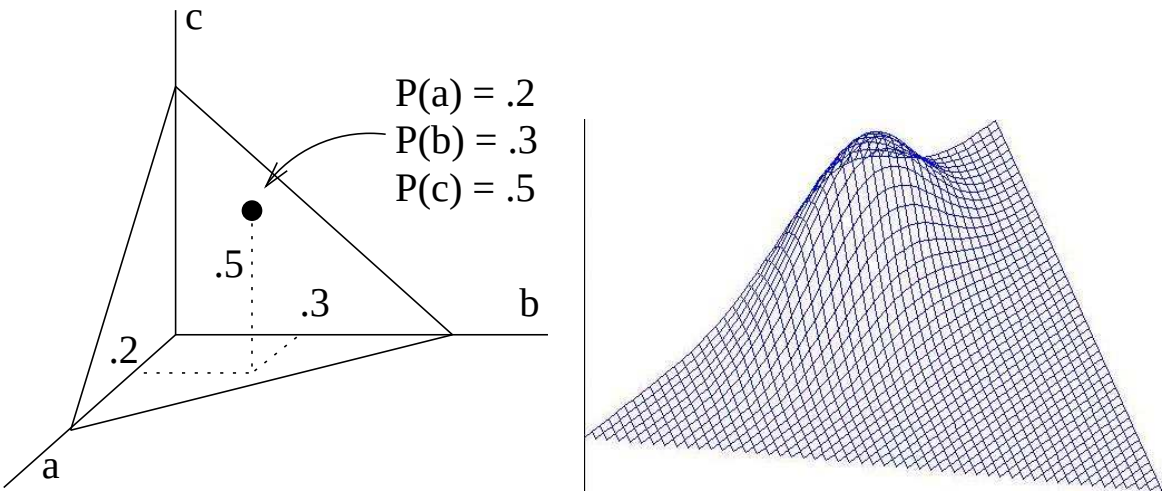

Figure 4-1: An example simplex for a multinomial that can take three different values (a,b,c). Each point on the simplex corresponds to a valid multinomial distribution; the Dirichlet distribution places a probability measure over this simplex. The second figure shows a probability distribution placed over this simplex.

is, if $\sum_i^m p_i$ does not equal one. The expected values of the Dirichlet distribution are given by

$$E[p_i|\underline{\theta}] = \frac{\theta_i}{\sum_j^m \theta_j}, \tag{4.1}$$

and the mode is

$$E[p_i|\underline{\theta}] = \frac{\theta_i - 1}{\sum_j^m \theta_j - m}. \tag{4.2}$$

As the conjugate distribution to the multinomial, Dirichlet distributions have the additional property that they are easy to update. For example, suppose we are given a set of observation parameters $\theta_1 ... \theta_{|O|}$ corresponding to a particular s,a. If we observe observation o_i, then a Bayesian update produces new parameters $(\theta_1, \ldots, \theta_i + 1, \ldots, \theta_{|O|})$. Thus, we can think of quantity $\theta_i - 1$ as a count of how many times observation o_i has been seen for the (s,a) pair. Initially, we can specify an educated guess of the multinomial distribution—which we take to be the mode of the distribution—and a pre-observation total that represents our confidence. Given a total number of pre-observations, we set the beta parameters of the Dirichlet distribution in proportion to the expected probability values for each state.

4.2 Approach

Table 4.2 describes our general approach. We have already described how we may put priors over the parameters; here we discuss how to solve for a policy given uncertain parameters, how to update the parameters, and how to update the policy.

Table 4.1: Expected Value approach to solving an uncertain POMDP. We start with distributions over uncertain parameters and refine them over time.

EXPECTED VALUE POMDP

- Put priors over all of the parameters
- Solve for an initial dialog manager policy
- Loop:
  - Interact with the user.
  - Update the distribution over parameter.
  - Update the dialog policy.

4.2.1 Solving for the Dialog Policy using Expected Values

The Q-functions in the Bellman equations described in Chapter 2 can be written in the following form:

$$\begin{aligned} Q(b,a) &= \max_i \vec{q_a} \cdot b, \\ q_a(s) &= R(s,a) + \gamma \sum_{o \in O} \sum_{s' \in S} T(s'|s,a)\Omega(o|s',a)\alpha_{n-1,i}(s). \end{aligned}$$

The first equation is an expectation over our uncertainty in the true state (in our case, the user's intent). The second equation averages over the stochasticity in the model: the reward we expect to get in a certain state is the current reward plus an average over all the future rewards we may get depending on which belief state we may transition to.

Computing the vector $\vec{q_a}$—which is an average over the stochasticity in the user model—now requires an additional expectation over our uncertainty in the user model. Let Θ represent collec-

tively all of the hyper-parameters of the distributions over the rewards, observations, and transitions. Then we can write the additional expectation as:

$$\begin{aligned} q_a(s) &= E_\Theta[R(s,a) + \gamma \sum_{o \in O} \sum_{s' \in S} T(s'|s,a)\Omega(o|s',a)\alpha_{n-1,i}(s)] \\ &= E_\Theta[R(s,a)] + \gamma \sum_{o \in O} \sum_{s' \in S} E_\Theta[T(s'|s,a)\Omega(o|s',a)\alpha_{n-1,i}(s)] \\ &= E_\Theta[R(s,a)] + \gamma \sum_{o \in O} \sum_{s' \in S} E_\Theta[T(s'|s,a)]E_\Theta[\Omega(o|s',a)]\alpha_{n-1,i}(s), \end{aligned}$$

where $E_\Theta[R(s,a)]$, $E_\Theta[T(s'|s,a])$ and $E_\Theta[\Omega(o|s',a)]$ are the means of the Dirichlet distributions as given by equation 4.1. The second line follows from the linearity of expectations, and the third line follows from the fact that the uncertainty in the transition and observation distributions—at least, in how we choose to model them—is independent. The $\alpha_{n-1,i}$ is a fixed value from the previous iteration and does not require averaging.

Note that learning does not appear in this derivation. What we have shown is that if the parameters are uncertain, for the optimal dialog policy—that is the policy that maximizes the expected discounted reward—it is sufficient to solve the POMDP with the expected values of the model parameters. This policy is not optimal if we can gain additional information about the parameters through our interactions. We will explore this issue in future chapters.

4.2.2 Updating the Parameters after an Interaction

Given a history of states, actions, observations, and rewards, it is very straightforward to update the priors over the parameters. For the rewards, which have Gaussian priors, we simply compute:

$$\mu'_{R(s,a)} = \frac{\mu_{R(s,a)} n_{R(s,a)} + r}{n_{R(s,a)} + 1} \tag{4.3}$$

$$\sigma'^2_{R(s,a)} = \frac{n_{R(s,a)}(\sigma^2_{R(s,a)} + (\mu_{R(s,a)} - \mu'_{R(s,a)})^2)}{n_{R(s,a)} + 1} + \frac{(r - \mu'_{R(s,a)})^2}{n_{R(s,a)} + 1} \tag{4.4}$$

$$n'_{R(s,a)} = n_{R(s,a)} + 1, \tag{4.5}$$

where $n_{R(s,a)}$ is the previous observation count, and $\mu_{R(s,a)}$ and $\sigma^2_{R(s,a)}$ are the previous mean and variance for the reward $R(s,a)$.

For the Dirichlet priors over the transition and observation probabilities, we simply increment $\theta_{o,s,a}$ or the $\theta_{s',s,a}$ as appropriate. Recall that the initial beta values can be thought of as pre-observation counts of a particular event; as we get true observations of the event, we simply add these observations in. As the beta values get higher, the priors over the observation and transition distributions will get more sharply peaked. In the case that we receive not a single observation, but a distribution over observations (such as normalized word counts from an n-best list of possible phrases), we simply update each beta value with the probability (which we can think of as a partial count) that it was observed: $\theta'_{o,s,a} = \theta'_{o,s,a} + P(o|s,a)$.

While the updates above are simple, a key assumption that we have made is that we know what state the system was in when a particular reward, observation, or transition occurred. (Note that our history, in a POMDP, consists of only actions and observations.) In general, this issue can be a very tricky problem. One approach—and the approach that we use—is to note that once we have a history of observations and actions, we have reduced our POMDP to a Hidden Markov Model (HMM) in which the observations and transitions are conditioned on the (known) sequence of actions. We used the Viterbi algorithm (see [31] for a good tutorial) to determine the most likely sequence of states for that HMM, and thus for that observation history.

We have glossed over one more point: when using the Viterbi algorithm, the system expects transition and observation distributions for each state. However, we do not know the true observation and transition distributions! We use the expected values of these distribution in our algorithm. In general, updating the distribution parameters with the HMM output could have fairly poor performance, since we may update the wrong parameters. Indeed, as seen in Figure 4-2, we see that in our basic dialog model, the number of mispredicted states grows steadily over time. However, most of the time, the error is that we fail to notice when a user changed their mind in mid-conversation. As a result, the values of the transitions become close to deterministic—that is, we begin to think that the user never changes their mind in mid-conversation, and we attribute the inconsistent phrases as additional noise in the observations. The resulting policy may be sub-

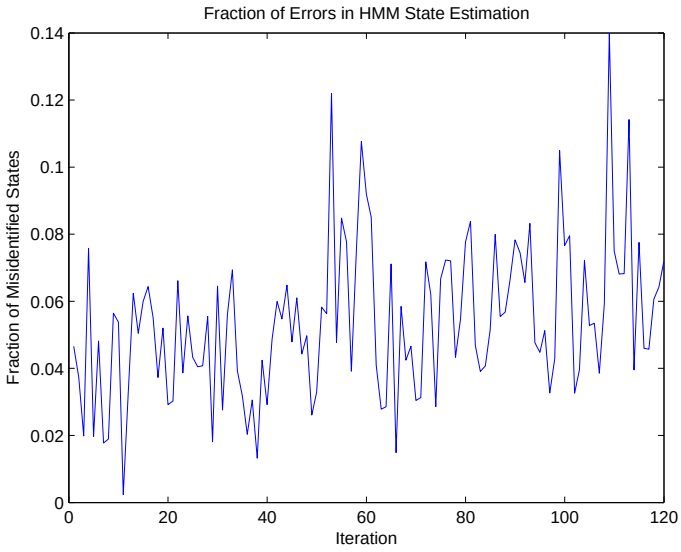

Figure 4-2: Error in the HMM estimation of the state.

optimal, but not critically, since we assume that the user is generally unlikely to change their mind in mid-dialog to begin with; we get approximately the same results by determining the user's true goal state by the motion action that successfully ended the dialog. Note that we use the cue that the dialog has ended as a very important cue for training.

4.2.3 Updating the dialog policy

The policy that we found in Section 4.2.1 gave the optimal course of action for a POMDP with certain parameters—the expected values of the distributions over parameters. Once we have completed a dialog and updated the parameter distributions, we have a new set of expected values. Clearly, we should consider changing the policy to reflect the new parameters (note this is a simple form of learning: although the system is not aware that its actions will result in greater knowledge about its environment, once that knowledge is obtained we do adapt to our new conditions).

One option would be simply to recompute the solution to the expected value POMDP each time the parameters are updated. This is undesirable, however, because solving the basic POMDP (in Matlab) can require several minutes of computing. Instead, we use the fact that the value function has probably not changed very much in the span of one interaction. Thus, we can use our current

solution as a starting point and compute additional backups (recall Equation 2.3). Since the backup operation is a contraction ([10]), the additional backups will always bring the old solution closer to the new solution.

The question remains of how many backups to perform, and in this work we consider three different update heuristics:

1. Backup to convergence. After each completed dialog, we perform enough backups for the new value function to converge.[2] This should lead to the best expected performance given the uncertainty in the user model. However, the current model statistics may come from a few unrepresentative dialogs so that the computed policy is wrong. Here, more learning must be done before the policy's performance can improve, and careful planning may therefore be wasted effort. Computing a POMDP policy to convergence is also a slow process, leading to long latencies in dialog manager responses.

2. Backup k times. Perform k backups, regardless of how the parameters change. This approach may prevent latencies in the dialog manager performance, but does not give the dialog manager time to compute better plans once we have confidence in the model.

3. Backup proportionally to model confidence. The sum of the variances on each parameter is a rough measure of the overall uncertainty in our user model. If an update reduces the overall variance, we backup $\lfloor k * dvar \rfloor$ times, where $dvar = \sum_{minM} \max(0, \sigma^2_{m,i} - \sigma^2_{m,f})$ where M is the set of all model parameters and $\sigma^2_{m,i}$ and $\sigma^2_{m,f}$ are the initial and final variance for model parameter m. Thus, $dvar$ measures the total reduction in variance. The intuition is that we wish to expend the most computational effort when the user model becomes more certain. For simulation purposes, we also capped the number of backups per update at 50.

The first heuristic, backing up to convergence, is the most principled, but it fails to capture the fact that if the parameters have changed only slightly, large amounts of computation may be a wasted effort. On the opposite end of the spectrum, backing up a fixed number of times is clearly suboptimal since there may be times when we really wish to

[2] To complete our simulations, we capped the number of backups per update step to 50.

We suggest the use of the final heuristic, at least in the initial planning stages, since it expends more computational effort when there is a large decrease in variance (note the $dvar$ is the sum of the change in variance for all the parameters). In later stages, when the parameters are already certain, it may be prudent to simply solve the new POMDP to convergence. That way, we avoid the issue of never applying backups because the variance increases very gradually; although we never had this issue in practice.

4.3 Performance

We tested our approach in an artificial simulation and with a robotic wheelchair (see Table 4.2 for a summary of the key parameters).

Table 4.2: Model Parameters for Simulation and User Tests.

Parameter	Simulation	User Test
States	7	7
Actions	12	12
Observations	11	19

4.3.1 Simulation Performance

Table 4.3 shows initial parameter guesses and true values. Initially, the expert prior believed that the voice recognition system was more accurate, and that the user was more forgiving, than the true values. The (slightly strange, obviously constructed for this experiment) user also preferred automated telephone-style interactions where the system listed a bunch of options as opposed to open questions about where they wished to go. (Granted, this is inexplicably odd behavior, but reasonable if the voice recognition was generally poor and thus able to distinguish yes/no commands much better than place locations.)

The model began with symmetric observation and transition probabilities, that is, we specified the probability of the most likely option, and the remaining options were uniformly distributed with the remaining probability. While not necessarily true in real world scenarios, it was a reasonable

Table 4.3: Initial and True Parameter Values for Simulation Tests. The initial model parameters assume a higher speech recognition higher accuracy and a user unforgiving of confirmation actions. This is an example of a very demanding user relative to the initial model specification.

	Initial	True
P(self-transition)	.95	.95
P(correct obs if ask-which)	0.7	0.5
P(correct obs if confirm)	0.9	0.7
R(complete task)	100	100
R(ask-which)	-1	-10
R(correct confirm)	-1	-1
R(incorrect confirm)	-10	-2
R(incorrect destination)	-50	-500

Table 4.4: Mean update times for each of the four approaches. Note that updates take place only after a dialog has been completed; when the dialog policy does not require any updates the dialog manager's average response time is 0.019 seconds.

Approach	Time (sec)
Convergence	135.80
1-backup	13.91
0.10-var	42.55
0.01-var	18.99

starting point and a model used in other dialog management systems ([39]). We attributed two pre-observations to each event to express a relatively low confidence in the initial parameter estimates.

To isolate the effect of our approach, we first ran the tests with an oracle that, once the dialog was complete, provided a complete state history to the dialog manager, eliminating the need to use an HMM to derive this history during learning. We note that the state oracle was not used during policy execution; during a user interaction the dialog manager chose actions solely from its POMDP model. Figure 4-3 shows the results averaged over 100 trials. All of the approaches performed similarly, but planning proportionally to the variance reduction achieved that result with almost as little computation (see Table 4.4) as replanning once per update. The variance reduction approach allowed us to focus our replanning near the beginning of the tests, when the parameters were most in flux.

What is also interesting is that the additional backups have a significant change in the variability of the solutions. Figure 4-4 shows box-plots of the interquartile ranges (IQRs) of the solutions. An

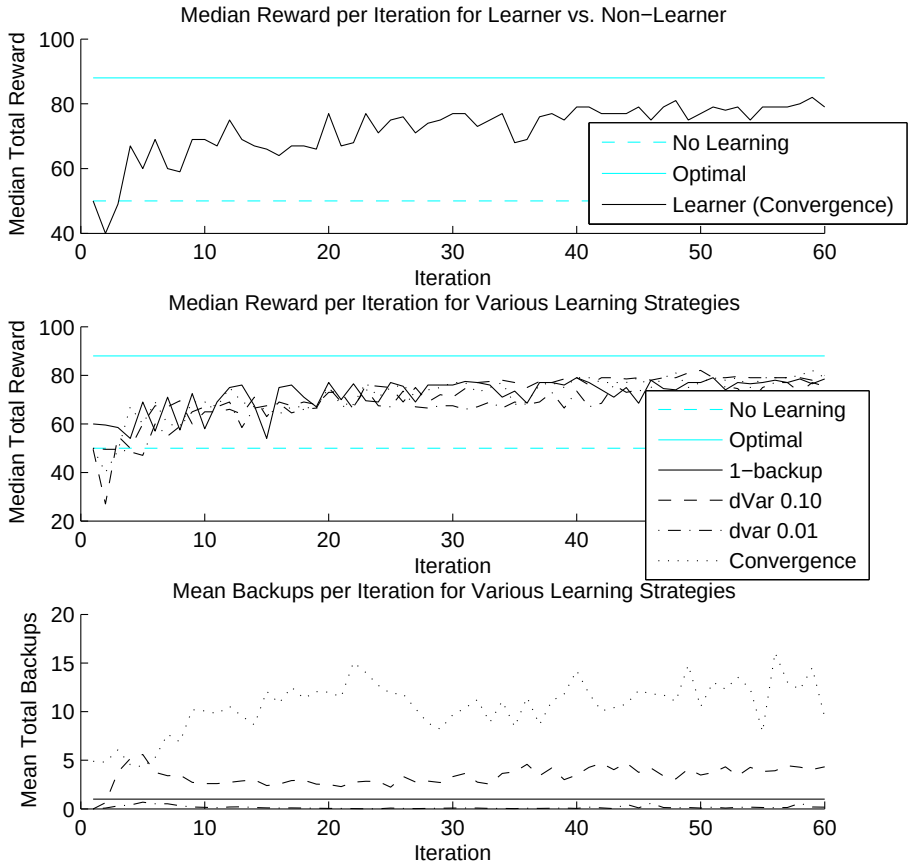

Figure 4-3: Performance and computation graphs. The learner outperforms the non-learner (top graph), and all of the replanning approaches have roughly the same increase in performance (middle graph), but replanning proportionally to the confidence of the model achieves that performance much less computation (and therefore faster response) than replanning to convergence (bottom graph).

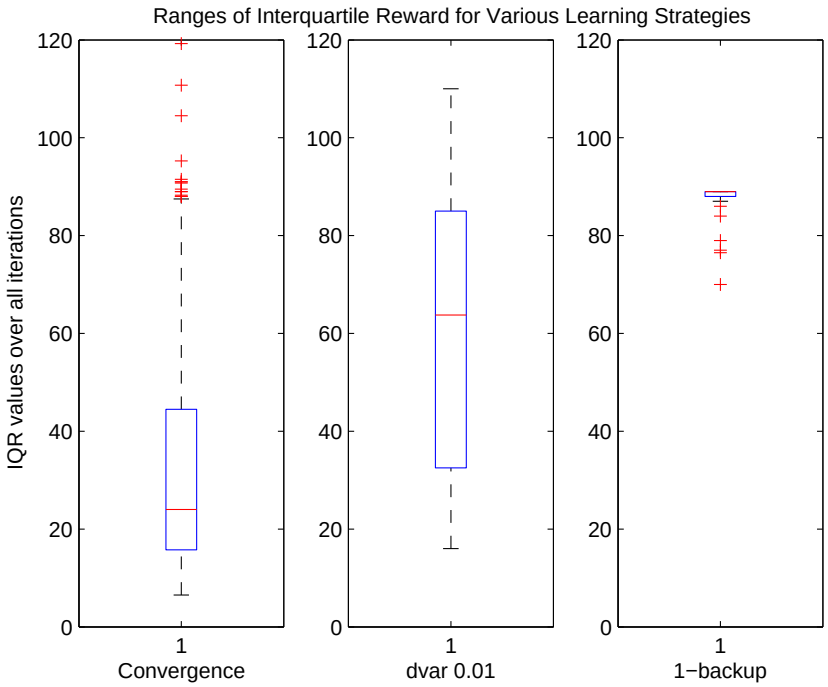

Figure 4-4: Interquartile Ranges (IQR) of the rewards. All of the replanning approaches have roughly the same median performance, but additional planning results in a more stable solution. Note that an IQR of 0 corresponds to zero variation around the median solution.

IQR of 0 would mean that the policy always resulted in the median value solution. As expected, iterating to convergence improves the stability of the solution; of the learners it has the smallest IQR range. However, it is interesting to note that even with just a few more backups, we can still get a policy with a lot less variation.

Finally, we did the same tests using the HMM instead of the oracle to determine the state history for the purpose of learning. For the reasons described in the previous section—with higher perceived observation noise, we followed a more conservative policy—the HMM performed slightly worse than the system with the oracle, but the difference was not significant given the noise in the system.

4.3.2 Wheelchair Performance

The next set of tests were performed on the wheelchair itself with the author as the user. To allow the experiments to run in real time, only one backup was computed after each dialog; the goal was

Table 4.5: Initial and True Parameter Values for User Trial.

	Initial Value	True Value
P(correct obs if ask-which)	0.6	see graphs
P(correct obs if confirm)	0.8	1.0
R(complete task)	100	100
R(ask-which action)	1	-30
R(correct confirm)	10	-1
R(incorrect confirm)	-10	-10
R(incorrect destination)	-200	-500
R(incorrect no-action)	-1000	-100

not to show speed of different planning techniques but to show general dialog improvement with model parameter learning. Although the wheelchair was capable of driving the various locations, we did not execute the motion commands for the purposes of the test. Table 4.5 shows the initial and true parameters for the user test.

At the time of the tests, we were using the Sphinx-2 voice recognition system [32]. The recognition quality was incredibly poor (in what seems to be a common complaint among Sphinx users; apparently one has to tune many internal parameters in the source code to get reasonable performance): in many cases the system failed to recognize anything in the utterance. In order to expedite the tests, we first spent a long time talking to the speech recognizer and collected the mistakes that it generally made. Next, we entered dialogs as text into the dialog manager. These dialogs contained common errors that we had observed in the voice recognition software as well as words and situations that we particularly wished to test for. In particular, the system contained:

- Speech Recognition Errors. Sphinx often mistook similar sounding words; for example, the software tended to mistake the work ‘desk’ with ‘deck.’ In the absence of this knowledge, however, we had assumed that we would observe the word ‘desk’ more often if the user was inquiring about the information desk and ‘deck’ more often if the user was inquiring about the parking deck. We also made difficult to recognize words more likely to be dropped (for example, ‘parking’ and ‘information’ were harder words for our software to recognize). Note that the speech recognition errors the system encountered were filtered to eliminate interactions where the recognition failed to produce any recognized utterances, and so these results do not

precisely capture all speech recognition errors of our current system.

- Mapping New Keywords. General keywords, such as ‘tower,’ or ‘elevator,’ were not initially mapped to any particular state. Some unmapped keywords were more likely in one particular state (such as ‘parking deck’ or ‘the deck’ for ‘parking lot’; ‘the elevator’ for ‘the Gates elevator’), while others occurred in several states (‘I’d like to go to the information desk a lot’ uses the word ‘lot’ outside the context of ‘parking lot’).

- Spurious Keywords. Especially in a spatial context, users could refer to other locations when describing the desired state. For example, they might say ‘the elevator by the information desk’ or ‘the Gates side booth.’ Our simple bag-of-words approach to speech analysis precluded complex language understanding; the dialog manager had to associate higher noise with keywords that occurred in many states.

For the user tests, we expanded our observation model to a total of 19 allowed keywords. Five of those keywords were pre-mapped to particular goal states. For example, we guessed that if the user wished to go to the Gates Tower, then the probability of hearing the keyword “Gates” would be relatively high (60%). Similarly, if the user wished to go to the Dreyfoos tower, we guessed that the probability of hearing the word “Dreyfoos” was also high (again, 60% in our tests). The remaining key words were not mapped to any particular state. For example, we initially guessed that the word “tower” was equally likely in any of the five goal states. Our goal was to have the system learn the mappings—if any—for these keywords depending on the vocabulary choices of the user. In the case of the “tower” example, if the user often used the word “tower” when referring to the “Gates Tower,” then we would learn that association. However, if the user never used the word “tower,” then that keyword would remain unmapped.

there was no longer one word associated with each goal state. Instead, we initialized the priors with a goal word per location—for example, ‘Gates,’ for Gates Tower. Other key words, such as ‘Tower’ or ‘Elevator,’ were left unmapped. There were a total of 19 possible observations. Our goal was to learn the vocabulary that the user tended to apply when in a particular state. We began by thinking that we would hear the keyword we had associated with each state about 60% of the time if we were in that state.

The observations were analyzed in several steps. First, we counted the presence of each key word in the output of the speech recognition system. If no keywords were present, we ignored the input and awaited another user response. The reason for only searching for keywords was to first to simplify the analysis; however, it also protected us from reacting to noise and other partial inputs to the system. Note that utterances rejected at this stage never made it to the POMDP planner, and thus our planner was not burdened with having to learn simply to ignore the significant fraction of supposed utterances that were actually noise. This also protected the planner from 'learning' that noise was the predominant quality of a particular state.

A normalized vector of counts was submitted to the dialog manager and incorporated into the POMDP as probability distribution over observations. (Note that in this case, it is not a probability distribution over what observations may have occurred, rather, it provided us a way to fold in multiple observations into the belief update step according to their weight.) If given a vector of observation probabilities $P(o)$, we simply extend the belief update equation to be:

$$b_n(s) = \eta \sum_{o \in O} P(o)\Omega(o|s', a) \sum_{s \in S} T(s'|s, a)b_{n-1}(s) \tag{4.6}$$

Since POMDPs deal only with maximizing the expected reward, this additional expectation does not change the behavior or validity of the planning process.

For the purposes of a fair test, the states were requested in the same pattern to the learning and the non-learner. The same set of phrases were also used when responding to the system when asked the open question "where do you want to go?" Also, the user never changed her mind about where she wished to go in the middle of a dialog, which meant that the HMM-based state prediction was exact.

Figure 4-5 shows that the learner generally performed better than the non-learner. The first dip in the plot, near interaction 10, is where the learner first encountered misused keywords, the hardest situation for it to deal with. The plot, which shows a single user test, also highlights a trade-off that was not immediately apparent in the aggregated plots from the simulation results. In the simulation results, we could happily initialize our pre-observation count to a very small number to indicate our uncertainty, and as a result, our learning rate was very fast. The aggregated rewards

smoothed over pitfalls in any particular run.

As we see by the dips in the plot (and this is after tuning the pre-observation count up from two to five, a number that yielded a slower but more robust learning rate), our learner makes mistakes. Often the mistakes, especially the mistakes made in the early stages, occur because the system gets too sure about certain distributions while others are still highly variable. For example, suppose that the system hears the word 'Forbes' during an initial dialog when the user wants to go to the Forbes cafe. If the initial pre-observation count is low, the system will suddenly become very confident that the word 'Forbes' is extremely likely if the user wants to go to the Forbes cafe. If system hears the word 'Forbes' a second time, it may choose to proceed directly to the cafe without even confirming the location with the user. Such a quick response may be fine if the user truly wants to go to the cafe, but suppose that the word "Forbes" was recognized in error in the second dialog. Without time to calibrate to the observation noise, the system will rush the unhappy user to the cafe! For more graceful learning, we must increase the pre-observation count; this way, the distributions do not peak too quickly during the initial phase. However, this means that more observations will be required to overcome biases in our prior.

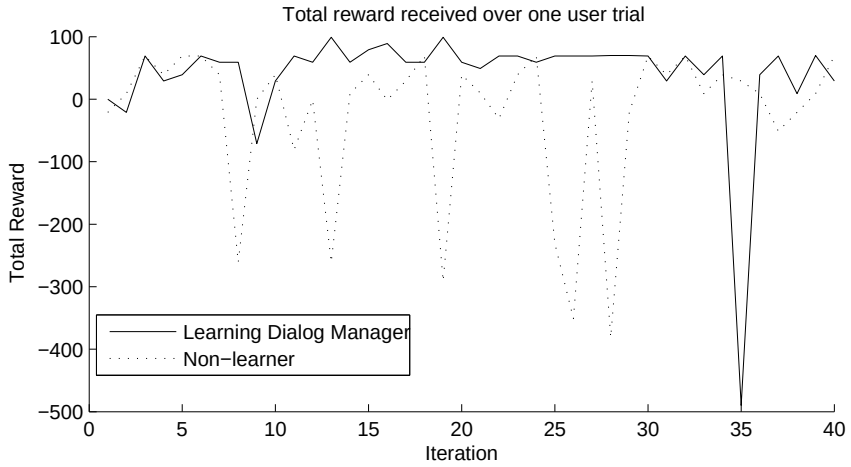

Figure 4-5: Policy-refinement based on expected values for a dialog with five goal locations from a single trial. The user was much less forgiving of mistakes than the dialog manager initially expected, and often used keywords that were not mapped to goal locations in the initial model. The learner (solid) generally outperforms the non-learner (dashed), only making one serious mistake (far right).

Despite these issues, the planner does reasonably well (and performs very quickly). To see how it handles various types of problems, we show its performance by state. Table 4.6 shows the

Table 4.6: Mean overall and per state results for the user test consisting of 36 dialogs. The learning dialog manager improves over the initial, hand-crafted policy in most states. The exception (State 3) is due to a single outlier.

	Overall	S1	S2	S3	S4	S5
Non-learner	-16.9	4.6	10.7	39.0	49.2	-187.7
Learner	37.2	54.6	50.4	-35.8	67.3	49.1

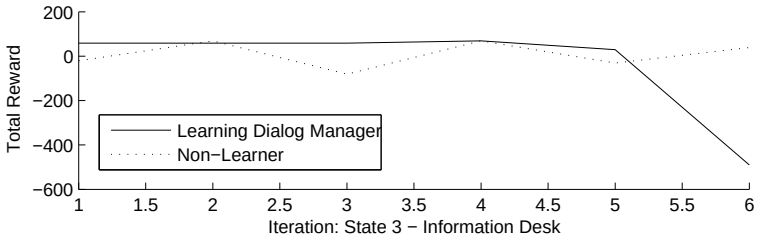

Figure 4-6: This plot includes only Information Desk requests. Observations in this state were 31% original keywords, 46% new words, and 23% misused keywords. Unmapped words more strongly associated with other states made it tough for the learner to improve the dialog policy.

average reward per state for the learner and the non-learner. In general, the learner did an excellent job of mapping keywords to states (regardless of how many states the new word was associated with). In these situations (see Figure4-7), the dialog manager was able to shorten interactions (and thus increase overall reward) by learning how to handle these new observations.

The dialog manager had the most difficulty when a state often had the occurrence of keywords that were highly likely in other states (see Figure 4-6). This was similar to the issue that occurred when the distributions initially became too certain—if a certain observation, such as 'Gates' was very likely to be see if the user wished to go the Gates Tower, the system was confused when it heard a phrase like 'go to that booth near Gates.' Since the observation distributions did reflect the true observation distributions, we believe that the central issue may have been not enough backups; one backup might have not been enough to learn the new policy of what to do when one has a very clean state and a very noisy state.

As a qualitative measure, Tables 4.7 and 4.8 shows how some of the dialogs changed due to the learning process. In the first pair, the user often uses the word 'deck,' which is unmapped, to refer to the parking lot. The system must wait until the word 'parking' is heard, but in the meantime, the user uses several keywords, such as Gates and Dreyfoos, that have already been mapped. This results in a rather painful learning experience. However, once the system adapts, it is able to handle the use of other words, such as elevator or tower which have since been mapped to other

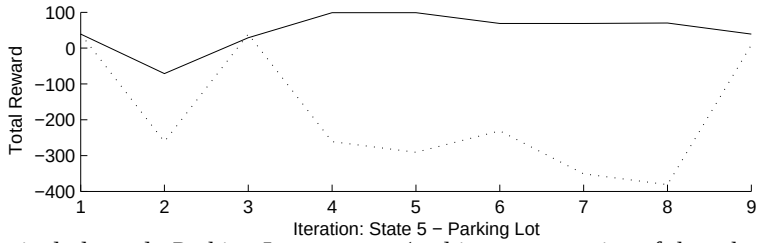

Figure 4-7: This plot includes only Parking Lot requests (and is representative of the other states). Observations in this state were 9% original keywords, 73% new words, and 18% misused keywords. The learner soon realizes that the user often refers to the parking lot with the unmapped work 'deck.' Since the user is forgiving to confirmation questions, the dialog policy applies them aggressively to disambiguate misused keywords.

locations, and the newly mapped word deck. The second dialog demonstrates two things. First, the system again learns a new keyword, elevator. It has also learned that although the building has several elevators, the user usually wants to go the Gates elevator. Therefore, it tries to confirm that location first.

Finally, as a second demonstration, we did another similar test, except this time we used phrases based on data from a different user speaking into the voice recognition system. This user was not trying to create particular obstacles, and as a result, we see that the effects of learning, while still evident, are less pronounced than before.

4.4 Discussion

The benefit of the expected-value approach is that it requires very little computational effort; we can easily learn both reward and observation parameters. The backups are fast enough that the dialog can be performed in real time. Also, this approach allows us to consider the parameters as continuous-valued numbers (as they should be): we do not need to discretized the parameter space. We also make no assumptions on what the final set of parameters may be: although the initial priors are symmetric, as we saw in the user test, there is no additional work required to learn distributions that are bi-modal (for example, if the word tower occurs with two locations, the Gates and Dreyfoos Towers).

The drawback to our approach essentially spends all of its time exploiting and none of its time exploring. Thus, even with an infinite number of trials, we cannot guarantee that the parameters

Table 4.7: Parts of dialogs between the user and the wheelchair. The dialog specification has failed to specify how to handle the words 'deck' and 'elevator.' In the first dialog, the non-learner is so confused when it first hears the word 'parking' that it takes a second utterance for it to select an appropriate action.

Without learning
USER: Take me to the deck between the two towers.
The user says "deck" to refer to the parking lot. The system does not know this yet.
ROBOT: Where did you want to go?
USER: Yes, I want to go to central elevator that goes to the deck
ROBOT: Where did you want to go?
USER: The middle deck, between Gates and Dreyfoos
ROBOT: Do you want to go to the Gates Tower?
USER: No.
ROBOT: Going to the Dreyfoos Tower.
USER: No! I want to go to the deck!
The system still cannot associate "deck" with any destination.
ROBOT: Doing nothing.
USER: Take me to the deck.
ROBOT: Where did you want to go?
USER: The parking lot.
ROBOT: Where did you want to go?
USER: I said the deck.
ROBOT: Where did you want to go?
USER: Yes, the parking deck.
ROBOT: Going to the parking lot.
After learning:
USER: Take me to the deck between the two towers.
ROBOT: Where did you want to go?
USER: Yes, I want to go to central elevator that goes to the deck
ROBOT: Do you want to go to parking lot?
USER: Yes.
ROBOT: Going to parking lot.

Table 4.8: In the second dialog, the learner realizes that while elevator refers to multiple locations, the user most often uses in the context of the Gates Tower.

Without learning
USER: Take me to the elevator.
The system does not know what 'elevator' means.
ROBOT: Where did you want to go?
USER: The Gates elevator please.
ROBOT: Do you want to go to the Gates Tower?
USER: Yes.
ROBOT: Going to the Gates Tower.
After learning:
USER: Take me to the elevator.
ROBOT: Do you want to go to the Gates Tower?
USER: Yes.
ROBOT: Going to the Gates Tower.

Table 4.9: Mean overall and per state results for a single end-to-end user test consisting of 36 dialogs. In general the learning dialog manager showed slight improvements, especially in mapping unmapped keywords (in this case, State 3).

	Overall	S1	S2	S3	S4	S5
Non-learner	45.0	69.0	59.0	-11.0	59.0	49.0
Learner	55.9	72.4	66.3	32.5	52.6	55.9

will eventually converge to their true values. In fact, it is easy to construct an initialization of a prior that will result in a policy that will never converge to the true optimal. For example, consider a scenario in which there are only two actions: *ask* and *confirm*. Suppose that under some prior belief p over reward parameters, we have the following relationship between the true rewards and their expected values under the prior:

$$R_{ask} > R_{conf} = E_p[R_{conf}] > E_p[R_{ask}], \tag{4.7}$$

where R_{ask} is the reward for asking a general query and R_{conf} is the reward for asking a confirmation question. If the dialog manager attempted action *ask*, it would discover that its belief about R_{ask} was incorrect. However, if the dialog manager only makes decisions based on the rewards it expects to receive, $E_p[R_{conf}]$ and $E_p[R_{ask}]$, it will never try the action *ask*. Thus, the dialog manager will be stuck with a suboptimal policy. This situation will occur if the domain expert estimates the reward means incorrectly, even if the expert states that he is very unsure about some of the values he chose.

In the next chapters we take steps to resolve this issue by incorporating the unknown parameters into the hidden state of the POMDP.

Chapter 5

Decision-Theoretic Approach

One of the primary concerns with the expected value approach in Chapter 4 is that it was not aware of the uncertainty in the model, and this made it possible for the system to get caught in local optima. Unaware of the risk, it also acted too aggressively at the beginning, when the parameters were not certain. In this chapter, we take the first step to resolving this issue by incorporating the unknown parameters as additional hidden state in the model. First (Section 5.1), we attack the problem assuming that the parameters are discrete. In Section 5.2, we show how we may consider continuous models.

5.1 Discrete Approach

While there exist extensions of the value-function based approach described in Section 2 for solving POMDPs with continuous state [29], they are fairly complicated. Thus, we begin by limiting the parameters to have a discrete set of values. In this section, we also restrict ourselves to learning only the reward parameters. We further assume that the rewards are symmetric and oblivious with respect to the states—that is, a penalty for taking an incorrect movement is the same regardless of what state the user is currently in and whatever state we try to drive to. Finally, we assume that the user will provide explicit reward feedback at each step (as in the standard model of reinforcement learning).

Table 5.1: Discrete reward values considered for the hidden reward parameters. The reward for a correct confirmation was -1 and the reward for a correct movement was 100.

R(general query)	-10, -5, -2, -1
R(incorrect confirmation)	-20, -10, -5, -2
R(incorrect movement)	-300,-200,-100

5.1.1 Model Definition

As before, this section will work with the simple five-goal model that we introduced in Section 2. If we assume symmetric rewards, there are five unknown rewards in the model: (1) the reward for asking a general query (ie, where do you want to go?), (2) the reward for a correct confirmation (ie, asking do you want to go to the cafe when the user wants to go to the cafe), (3) the reward for an incorrect confirmation (ie, confirming the cafe when the user wants to go somewhere else), (4) the reward for a correct movement (ie, driving to the correct location), and (5) the reward for an incorrect movement (ie, driving to the wrong location).

Without any loss of generality, we can set two of these values to fix the absolute and relative scales of the reward values. For our tests, we set the reward of correct confirmation to -1 and a correct action to 100. This left three more reward values to learn. For each of these, we considered four possible discrete values for the parameters. Table 5.1 shows the values we considered for each of the remaining parameters. The observation parameters were assumed to be known and symmetric, with p_{ask} = 0.7 and p_{conf} = 0.9. Since the policies are fairly robust to the actual reward parameter values, we felt that even this coarse discretization spanned a number of reasonable policies.

Figure 5-1 shows how adding the reward parameters as additional hidden state change our POMDP. Now, the state consists of a vector $\{s_u, \vec{s_r}\}$, where s_u is the user state (where they wish to go, same as before) and $\vec{s_r}$ is the preference state (what reward values satisfy the user's preference model). We assume that the reward values are stationary over time, thus the only state that changes is the user state. The preference state is fixed but hidden. All combinations of reward values and user states are possible, for a total of 336 states.

We require the user to give a reward feedback at every step. We assume that this is not terribly noisy, as it is likely to be a button press; for our simulations we let the confidence in the reward

value be equal to p_{conf}. We extend our observation space to be $\{o_d, o_r\}$, where o_d is the speech to the dialog manager and o_r is a reward entered by the user. Considering all the discrete reward values that we could observe, this model has a total of 72 observations (8 keywords and 9 reward values).

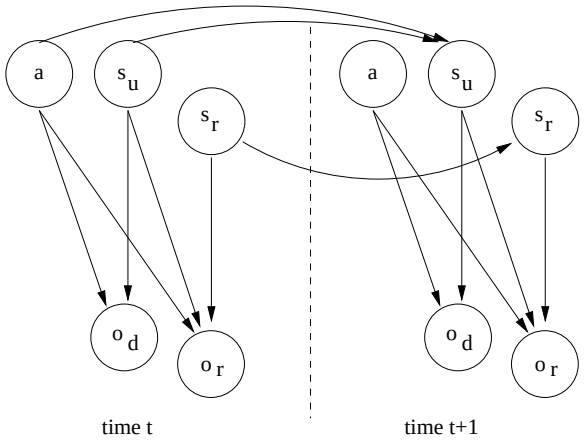

Figure 5-1: Arrows in the POMDP influence diagram connect elements that affect other elements. The robot's action a and the previous user state affect the current user state s_u, while the user's preferences s_r never changes. The observed reward o_r depends on the user's state and his preferences and the observed dialog o_d depends only on the user's state.

Unlike in the basic model, in which the POMDP was not aware of the reset (to speed convergence), in this parameter POMDP it is important to include the fact while we return to the start state once the dialog is done, we do not completely reset: our belief in the preference state does not go back to some initial prior once a dialog is complete. Thus, we retain the learning about preference states that occurred during previous dialogs.

5.1.2 Approach

In theory, since we have reduced our original problem of how to behave with uncertain parameters to a larger POMDP, all we need to do now is solve the resulting POMDP using the techniques that we have already discussed. In the new POMDP (the "parameter" POMDP), we have a belief b($s_u, \vec{s_r}$) that represents the probability that the user is in state s_u *and* the rewards are given by $\vec{s_r}$. Unfortunately, even though we started with a very simple model—five goal locations, three or four possible reward values for three parameters—we have a POMDP with 336 states, 12 actions, and

72 observations. Large POMDPs are difficult to solve, so we must take some care in how we create the dialog policy.

One reason why solving a large POMDP takes more time than solving a small one is simply a matter of computation. In each Bellman backup, we are multiplying matrices that have at least one dimension the size of the state space. Matrix multiplication is close to cubic in the size of the matrix, so increasing the size of the state space by a factor of about 50 increases the matrix multiplication time by 125,000. The factored nature of the model, however, can help us avoid some of those computations. For example, consider the probability of observing a particular $\{o_d, o_r\}$ in a state $\{s_u, \vec{s_r}\}$. The observed speech input does not depend on the user's reward model, so the observation probability factors as:

$$P(o_d, o_r | s_u, \vec{s_r}) = P(o_d | s_u) \cdot P(o_r | s_u, \vec{s_r})$$

Now, when computing a belief update, we can update b(s_u) independently of b($\vec{s_r}$). In the end, b($s_u, \vec{s_r}$) is the tensor product of b(s_u) and b($\vec{s_r}$). We can compute parts of the backup operation with similar tensor products since identical factorizations exist for the other transition and observation probabilities. Note that we can factor only because the each observation part gives information only about one part of belief. Also, note this is not the same as a factored POMDP, in which different parts of the state space have different actions (which is a much more powerful concept). Table 5.2 gives the flow of our approach; as described in Chapter 3, we continually resample beliefs to refine the accuracy of our solution around the regions that are most relevant.

One question we have not addressed yet is the initial choice of prior over the preference states (which is our initial b($\vec{s_r}$)). One option is to simply put a uniform prior over all of the preference states. However, note that given the properties of the POMDP, as long as we start b($\vec{s_r}$) with full support over all of the preference states—that is, the probability of each preference state is nonzero—we will eventually converge to the correct preference state. Thus, if we wish to be more robust at the beginning of the learning process, we can skew our initial prior toward harsher penalties. The resulting initial policy will be more conservative and make fewer mistakes while the prior converges (this is one way to get robustness while staying within the POMDP's expected

Table 5.2: Parameter POMDP approach to solving an uncertain POMDP. We assume that the set of possible rewards is discrete and rewards are observed at each step.

PARAMETER POMDP

- Choose a starting belief and sample an initial belief set.
- Solve the Parameter POMDP on that belief set.
- Loop:
  - Interact with the user.
  - Update starting belief set.
  - Sample a new belief set.
  - Update the dialog policy.

value approach).

5.1.3 Simulation Performance

Figure 5-2, mostly a sanity check, shows how the parameter POMDP, if initialized to the correct prior, performs just as well as a well initialized expected value POMDP. The goal was simply to show that our resampling approach does in fact do reasonable things, whereas with a poor initialization, we actually do unreasonable things if we are using the expected value approach. (Note: the expected value approach was initialized to mean values included the discrete set of rewards, but it tried to learn the rewards over a continuous space. This does not affect the issue with the algorithm, however.

In Figure 5-3, we show the results for the parameter POMDP for a variety of priors. In each case, the overall performance is about the same, showing (as we expect) that the parameter POMDP approach is robust to the initialization. What is most interesting, however, is that if we start out with a conservative prior, that is, a prior that puts most of its weight on a tough set of rewards, we do not see the initial dip in the learning process. Initially, the system is robust because it is very cautious of doing the wrong thing. In the end, it is robust because it has focused its samples on the important part of the belief space.

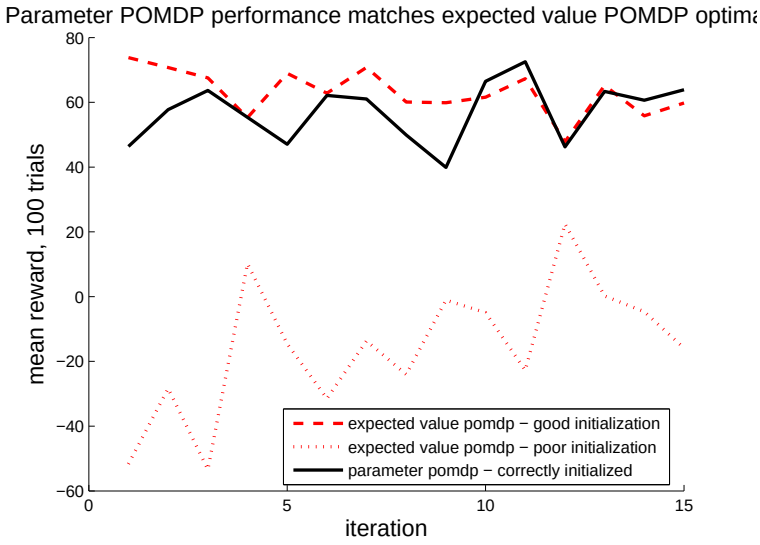

Figure 5-2: The figure shows the performance of parameter POMDP initialized with a good prior compared to different initializations of the expected value POMDP, aggregated over 100 simulated trials. The parameter POMDP, considering 48 possible user preference states, achieves the higher (correct) level of performance without sensitivity to initialization. The parameter POMDP seems to do slightly worse than the well-initialized expected value POMDP, but the difference is not statistically significant: on the final trial, the expected value POMDP has median 85 and IQR 16.5, and the parameter POMDP has median 86 and IQR 11. The poorly-initialized expected value POMDP reaches a median of 26.5 and IQR of 207 after 120 trials.

5.2 Continuous Model

In the previous section, computational power limited us to consider a few discrete reward values, and we were unable to do any learning on the observation model. In this section, we consider the case of only learning a continuous observation model. (We will return to learning both the reward and the observation model in Chapter 6.) As before, we will consider the unknown parameters as hidden state in the model, however, since those parameters now take on continuous values, we will use a sampling based approach to solving the parameter POMDP.

5.2.1 Model Definition

We continue to use the simple 5-goal model for the dialog, however, we now consider all of the observation parameters as additional hidden state. These observation parameters are fixed but unknown. We will consider two cases: in the first case, we assume that the observation model is symmetric, only that we do not know the true values for p_{ask} and p_{conf}. In the second case, there

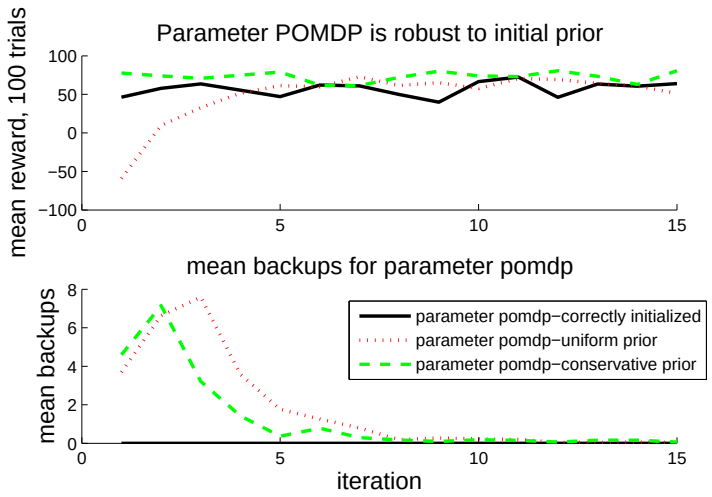

Figure 5-3: The three different priors—a uniform prior, a correct prior, and a conservative prior—converge to the same steady-state performance, but choosing a conservative prior allows us to do as well as if we had known the user's preference state initially. All solutions were backed up to convergence.

is no parameter tying and the observation distributions may take on any form. Because we use a sampling-based approach to solving the parameter POMDP, the only difference between these two cases is the form of our priors, the algorithm is otherwise identical. (Note that learning a full distribution will, of course, take longer than learning a symmetric distribution.)

5.2.2 Approach

Solving continuous state POMDPs is extremely difficult, so we use a sampling based approach to the problem (somewhat similar to [15]. We begin by sampling a set of POMDPs (between 15 and 50) from an initial distribution of parameters. Each one of these POMDPs is relatively small and therefore quick to solve; to further speed up backups, we can backup only a small fraction of the belief points and a small fraction of the alpha vectors in each iteration. Each POMDP begins with equal weight w_i. The discrete sample set of POMDPs approximates our continuous distribution over POMDPs. Each POMDP carries its own belief, approximating our distribution over states that we may be in.

One way to think of our small POMDPs is that each small POMDP value function represents a "slice" of the underlying continuous state POMDP value function where a set of parameter values

are fixed. While it is tempting to interpolate the value function at unsampled points based on combinations of the known slices, this approach does not lead to a good estimate of the underlying continuous state value function. Interpolating between sampled POMDPs is equivalent to giving a belief over those POMDPs. We know that the value of a belief is not weighted average of the values of being in each state (which is the Q_{MDP} heuristic), rather, we pay an additional price for the uncertainty.

Without a convenient way to approximate the value function, we turn to other approaches to choose the appropriate action. We apply a Bayes risk criterion:

$$a = \arg\min_a \sum_i w_i (Q_i(a, b) - V_i(b)) \tag{5.1}$$

where w_i is the weight of model i, $Q_i(a, b)$ is the value of taking action a in model i, and $V_i(b)$ is the value of being in belief b according to model i. Note that $V_i(b) = \arg\max_a Q_i(a, b)$, so the risk is always never positive. For each model, the term inside the sum measures how much we expect to lose if we take a particular action instead of the optimal action. Overall, the Bayes risk criterion states that we should take the action the minimizes the expected loss over all models.

By considering the potential loss, the Bayes risk criterion will prefer "safer" actions rather that actions that maximize the expected immediate reward. This caution is a desirable property because we are no longer solving the underlying continuous state POMDP and can only afford to do this one-step lookahead computation (the lookahead occurs in our computation of Q_i from V_i). Our actions are only looking one step ahead instead of many, and thus we choose a safer selection approach. We note that our action selection differs from Medusa [15]; Medusa chooses actions stochastically based on the weights of the POMDPs.

The second question we must answer is how we should update the POMDP set as we gain more information. As in the expected value case in Chapter 4, we can update the prior over the observation distribution after a dialog is completed. Even though we have the explicit distribution available, we can view our POMDP sample set as a set of particles approximating this distribution (which they do for the purposes of determining a policy). Thus, we can update the weight on each distribution based on the updated likelihood of that POMDP based on our prior. Suppose we

update our Dirichlet priors on distributions $\{\vec{\alpha_1} \ldots \vec{\alpha_k}\}$. Then the new weight for each model is:

$$w_i = w_i \prod_j^k f(\vec{p_j}; \vec{\alpha_j}) \tag{5.2}$$

where $f(\cdot; \vec{\alpha})$ is the probability density function for the Dirichlet distribution parametricized by $\vec{\alpha}$ and $\vec{p}$ is the sample POMDP's particular observation distribution. We normalize the weights to ensure that they always represent a valid distribution over the samples.

After each dialog, we also resample new POMDPs from the updated prior. For the present, we set the weight threshold to be $1/sqrtn$, where n is the number of POMDP samples. Replacing the POMDPs can take a significant amount of time, so we find quick ways to approximate the solution (note that since there are many POMDPs in our sample, we do not have to solve them all perfectly; the uncertainty in the parameter space makes it unnecessary to be able to draw fine distinctions based on a belief). In Chapter 3, we mentioned how we can exponentially decrease the number of beliefs required for problems with certain symmetries. We use that approach here for our first set of simulations.

Using ideas from Perseus as well as PBVI, another approach we use to decrease the time required to approximately solve the POMDP is to update only a small ($\sqrt{n}$) random fraction of the beliefs with only a small ($\sqrt{m}$) random fraction of the alpha vectors. Since the beliefs are multiplied by the alpha vectors several times, our overall computation time is reduced by a significant factor, which makes it possible to do the update several POMDPs in quasi-realtime (about 0.40 seconds per POMDP, instead of several minutes). The intuition behind sampling from the belief set is identical to Perseus: improving one belief may improve several beliefs around it. Moreover, nearby beliefs often produce similar alpha vectors with similar policies. Since our alpha vector set tends to be redundant, sampling from the alpha vectors produces reasonable results. We only need one of the redundant copies to be randomly selected for our backup to be approximately correct.

Since we are sampling such a small set, both from beliefs and from alpha vectors, we cannot just keep the set of updated alpha vectors as our new set; the chances are too high that we did not use one of the key support beliefs in a particular iteration, or that we did not select the particular alpha vector that matches a particular belief. As Perseus does with beliefs, we now keep around a

large set of potential alpha vectors. On each iteration, we do not remove the previous alpha vectors, we just keep adding more of them. If we exceed a desired number of alpha vectors, we can prune those from the initial set as those were the most likely to have been backed up and improved upon already. Note that checking several alpha vectors against one belief is still fast, however, so action selection can still be computed in realtime. Table 5.2.2 summarizes our approach.

Table 5.3: Sampling POMDP approach to solving an uncertain POMDP. The observations parameters are now continuous valued.

SAMPLING POMDP

- Sample a set of POMDPs from an initial prior.
- Approximately solve that set of POMDPs.
- Loop:
  - Choose the action that minimizes Bayes Risk.
  - Wait for an observation.
  - At the end of a dialog, update observation priors, reweight POMDP samples based on weights, and resample POMDPs with low weight.

5.2.3 Simulation Performance

There are several differences between our sampling-based approach and Medusa's approach. First, we replace all POMDPs with a weight below a certain threshold, not just the one POMDP with low weight. (We do not resample all of the POMDPs for computational efficiency; if a POMDP is performing well, there is no reason to replace it.) Second, we use Bayes risk to sample actions instead of picking actions stochastically. Finally, since our problem has natural breaks—the end of a dialog—after which it is easy to infer the previous state sequence, we do not have to specify a procedure for making oracle-queries. In this section, we show simulation results that demonstrate the effect of each of these factors.

We tested our approach using our basic five-state model with three unmapped observations. The prior distribution over POMDPs was initialized to believe that the probability of hearing a correct confirmation was 0.8 and the probability of hearing a correct state on a general query

was 0.6. The transition and observation probabilities were assumed to be symmetric, with the unmapped observations being extremely unlikely. We weighted our initial Dirichlet parameters to reflect approximately 10 total observations per state-action pair. We assumed that the user did not change their mind in mid-dialog.

The true POMDP had a -300 penalty for incorrect actions, a -1 penalty for questions, and -10 penalty for incorrect confirmations. At the initial transition, the probabilities of the goal states were $\{.32,.22,.16,.15,.15\}$. The observation probabilities for an "ask"-action were also asymmetric: in the first state, we were as likely to hear the first unmapped word as we were the original keyword. In the second and third states, we were almost as likely to hear one of the neighboring states as we were the keyword for that state. The final two goal states were as likely to here their keyword as the second unmapped keyword. The final unmapped keyword remained unmapped. The probability of getting a confirmation answered correctly was 0.8. We note that the asymmetric nature of the true transition and observation distributions is closer to what we saw in the initial wheelchair user studies and more realistic than a basic symmetric model.

Figure 5-4 shows the mean and median performance of the various approaches, all of which used the same set of beliefs. The solid gray line shows the performance of the "optimal" policy, which was found by doing PBVI-backups until the value function had converged. The dashed gray line shows the performance of the approximated policy, where 25 backups were performed using only a square-root fraction of the beliefs and alpha-vectors during each backup. Note that the approximation technique is fairly close to the optimal policy. The remaining curves show the performance of our sampling approach, our sampling approach using stochastic action selection (instead of Bayes risk), and the (basic) Medusa algorithm. All of the POMDP samples for these approaches were solved using approximate backups. The statistics are computed over 100 trials.

We see that after some initial noise, all of the learning approaches improve; however, using Bayes risk for action selection learns faster and appears to converge to close to the optimal solution. Our same strategy for resampling POMDPs but with stochastic action selection, shown by the dashed black line, performs about as well as the Medusa algorithm. Thus, the difference between Medusa's performance and our approach is not simply because we are more liberal with replacing

POMDPs with low weight. When we look at median performance, we see that using Bayes risk actually out-performs the optimal solution (recall that the optimal policy is trying to maximize mean performance). This indicates that our policy generally does well and suffers due to occasional large mistakes.

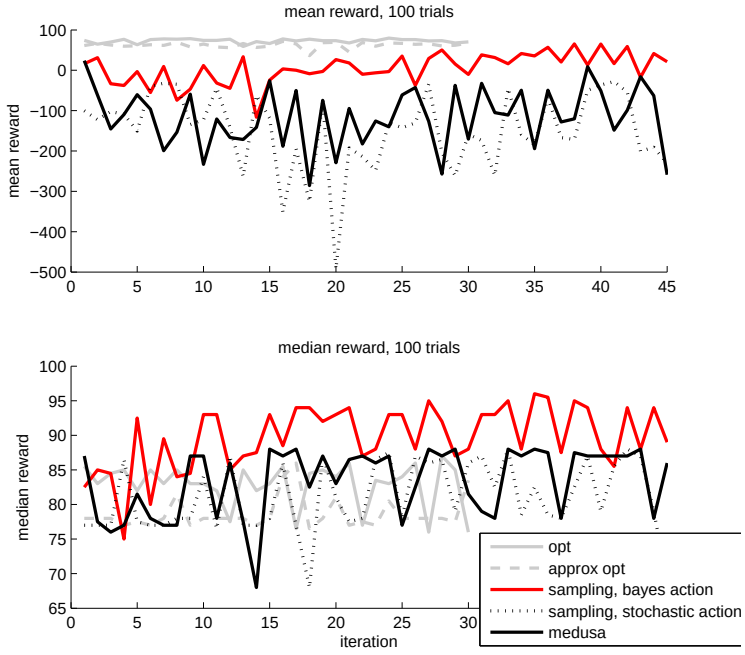

Figure 5-4: The top graph shows the mean reward (from 100 trials) achieved at each stage in the dialog. Neither of the learning approaches quite reach the optimum, but the action selection using Bayes risk has overall better performance. In the bottom graph, we see that our approach actually has median performance greater than the optimal, however, as we see in Figure 5-5, this performance in achieved at the cost of more severe errors.

Figure 5-5 shows the fraction of the 100 trials that failed on any particular dialog iteration. Here, failure means that the dialog manager decided to take the user to the wrong location at least once before taking them to the right location (the dialog manager rarely failed to eventually take the user the the right location). As expected, the approximation to the optimal policy fares slightly worse than the optimal, but the difference is not large. Both Medusa's and our proportion of errors decrease with time, and while neither reach the optimal level, our error rates are consistently lower than Medusa's. Thus, our approach not only achieves good average case performance, but we also are reasonably robust.

Finally, we note that using Bayes risk does require more computation that stochastic action

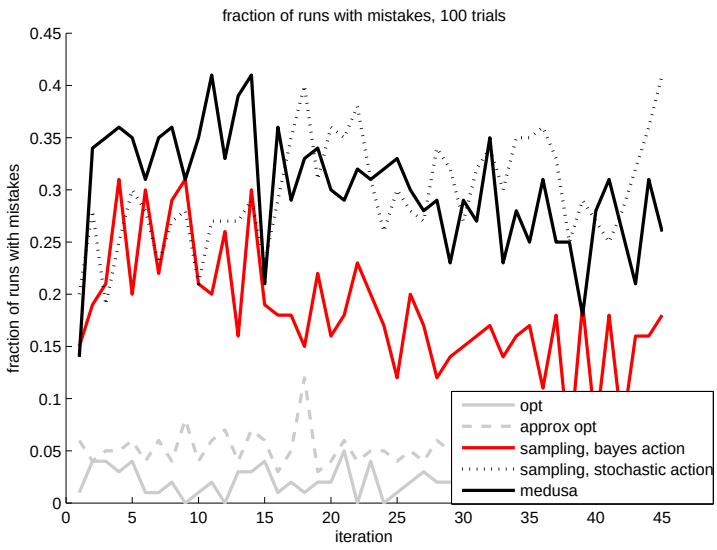

Figure 5-5: None of the learning approaches achieve the optimal error rate, although risk-based action selection does make fewer mistakes. Here, mistakes are going to the wrong location.

selection (essentially, the difference is between a zero and a one step look-ahead), but as it can be computed quickly, it seems like a clear win for the problem. One reason why we must be a little careful in supporting action selection using Bayes risk is that one of the benefits of Medusa's approach is that it guarantees convergence to the true problem parameters. While the Bayes risk criterion does well in our simple problem, we note that since we will visit all the states over the course of the experiments and experience the obervation distributions because of the simple queries available to use, we do not have to worry about parameter convergence. However, there may be situations where it does not perform as well; for example, if the POMDP believed a particular state or action's observation was so noisy that, given the cost, it was not worth visiting, it may always choose a more conservative alternative and never discover the true observation distribution for that probably useless state. (Note that since we do not have a deep lookahead, we may not realize the usefulness of exploration; Medusa forces exploration by stochastic action selection.)

5.3 Discussion

In this chapter we have discussed two approaches to incorporating parameter uncertainty into our model. In the first case, we choose a discrete set of possible parameter values and build a large parameter POMDP that treats the parameter value as additional hidden state. The result is a more cautious and robust learner: we find that as long as we initialize our prior over the possible models to have full support, the prior will converge to the correct model. Moreover, while the system is uncertain about which model it is in, it will behave more cautiously; we found that by setting a conservative prior we can make the system learn with fewer errors.

The main trade-off with using a discrete set of models is that our true POMDP must be one of the enumerated models, otherwise, we will be unable to converge to the true POMDP. In order to ensure that our true POMDP is part of the set, we may need a large set of models, especially if there are several parameters about which we have very little initial information. However, the size of our parameter POMDP is kn, where k is the number of models and n is the number of states in the model; for large values of k, solving the POMDP quickly becomes intractable. The problem is made even worse if we include the rewards as part of the observation; now we have increased the size of our observation set to $|O'| = |O||R|$. Recall that solving a POMDP is doubly exponential in the number of observations, in our approximations, this translates into the long loops over large matrices. To some extent, we alleviated the problem by solving the POMDP incrementally, and resampling beliefs reachable from the current belief.

The second part of this chapter we considered a continuous unknown observation parameter space. Thus we avoided the problem of trying to enumerate all possible models, but at the expense of no longer being able to approximate the value function at all. Instead, we sample a set of POMDPs from our prior over observation functions. Although we cannot do a deep look-ahead to determine what action to take, we show that by taking the action that minimizes the Bayes risk, we can behave robustly—our risk criterion makes us cautious—while still learning the true parameters.

We note that the best approach may lie somewhere between our discrete model approach and our sampling approach. In the discrete approach, a central question was how many models to use,

and we risked not including the true model. In the sampling approach, on the other hand, our model class was probably too expressive: in our particular problem, we are not really interested in fine distinctions; if we ask a general question instead of a confirmation when the expected rewards are approximately equal, it does not really matter if we chose the slightly suboptimal action. In reality, there are many fewer interesting policies than there are parameter initializations. Ideally we would like to sample from some set of ϵ-different policies instead of from the parameter or model space, however, this is beyond the scope of this work.

Chapter 6

Meta-Action Queries

In this chapter, we develop this work's final contribution toward more robust parameter learning. So far, our system's learning has been limited to its experience: it tries an action, experiences a negative reward, and learns not to do that action again. While explicit reward feedback is a standard aspect of reinforcement learning, this approach can be unsatisfactory for dialog management because the agent is forced to make a mistake to learn about its consequences. A similar problem can occur if a user repeatedly uses a foreign word to refer to a particular goal. Without the ability to ask about the meaning of the word, the system must wait, confused, until the user provides an explanation that it understands. Here we explore meta-actions, or actions that will help determine future actions, that the system can use to actively learn about the user's preferences and vocabulary choices.

Our first kind of meta-action query, the policy query, is a question about what the agent should do given its knowledge. For example, the agent might say, "If I'm 90 percent sure that I know where you want to go, should I just go instead of confirming with you first?" If the user says no, then the agent knows that the user places a high penalty on incorrect actions without experiencing its effects. Similarly, by asking a question such as "When I'm uncertain, should I list all the options I know (instead of asking you to repeat an option)?" we can find out what kinds of interactions the user prefers. The user's response to the meta-action allows us to prune inconsistent preference states from the belief space.

A secondary benefit of policy queries is that the agent is in charge of the learning process. The user does not have to repeatedly enter rewards to train the system; it is the agent that decides to ask for clarification when it is not sure about how to behave. A few meta-actions can quickly reduce the uncertainty in the user's preference state, so overall the user has to provide the system with less training-type input.

Another kind of meta-action query, the observation query, asks the user about the meaning of a new word. For example, the agent might ask an open-ended question such as "I know how to go to the following places: <list places>. Which one do you mean when you say kiosk?" It may also ask more specific questions such as "When you say the kiosk, do you mean the information desk?" Unlike policy queries, where no amount of listening will provide information about the user's reward preferences, one could always learn these word associations simply by observing the user for a period of time. An observation query only helps speed up the learning process. As we will see in Section 6.2, fast learning can be critical for success in dynamic, noisy situations.

Paralleling Chapter 5, this chapter will first consider the case of learning from a discrete set of models and then extend the analysis to a continuous space. In Sections 6.1 and 6.2, we consider policy and observation meta-action queries separately. In Section 6.3, we learn continuous reward and observation parameters using policy queries.

6.1 Discrete Approach: Learning Preference Models

In this section,[1] we will begin by studying the situation where the observation model is known. Just as in Section 5.1, we assume that there is a discrete and finite set of possible user preference models to be learned. However, the user no longer provides any explicit reward feedback; the only way the agent may learn about the user's preferences is to ask the policy meta-actions. We note that although the agent's policy depends on both reward and observation parameters, policy queries can only provide information about the user's internal reward model because the user is not aware of the observation noise in the system. We will address the issue of reward-observation coupling due to policy queries in Section 6.3.

[1] This work was previously presented in [6].

6.1.1 Model Definition

As before, we start with our basic five-goal model. We consider a scenario where there are four possible user types: the user may attribute either a -500 or a -100 penalty for incorrect movements and either -10 or -5 penalty for incorrect confirmations. For all user types, general queries cost -5, correct confirmations cost -1, and correct movements receive a +100. We chose these values to ensure that there were large qualitative differences in the policy, since in general the policy tends to be fairly robust to different reward values.

Figure 6-1 gives a qualitative feel for how the policies from the four different preference states differed. Along the x-axis are the indices for the four preference states: ($r_{move}, r_{confirm}$) = { (-500,-10), (-100,-10), (-500,-5), and (-100,-5) }. Along the y-axis are indices for specific user beliefs (not listed). The dots represent what type of action the agent should take in the corresponding preference state and user belief. Red dots indicate that the agent should ask a general question, yellow dots indicate that the agent should make some kind of confirmation, and green dots indicate that the agent should complete some kind of movement. Black dots indicate the robot should do nothing. As we expect, there are more green dots in preference states two and four, where the penalty for movement mistakes is low, and there are more red dots in preference states one and two, where the penalty for incorrect confirmations is high. By scanning across rows, we can discover what user beliefs are the most discriminative of the underlying preference state and therefore most useful for developing meta-action queries.

With only a small number of user types and a set of distinguishing user beliefs, we can finely tune the nature of the policy queries. From the perspective of our system, our goal is to find a user belief $b(s_u)$ where the policy given ($b(s_u)$, $\vec{s_r}$) depends on $\vec{(}s_r)$. Then, if we ask the user what we should do in $b(s_u)$, we can discover information about the user's preference state $\vec{s_r}$. Unfortunately, this kind of question may be non-intuitive for a non-expert user. If the number of possible preference states is small, we can phrase the meta-action query in ways that may make it easier for the user to understand and interpret. For example, the policy queries used in this scenario were:

1. If I am 90% certain of where you want to go, should I still confirm with you before going there?

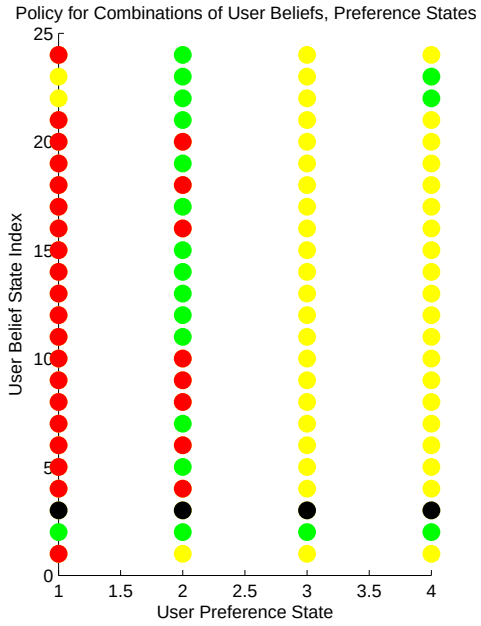

Figure 6-1: The dots indicate what action the robot should take if it knew the user's preference state (indexed on the x-axis) and had a belief over the user state (indexed on the y-axis). Roughly, the user states increase in uncertainty as one scans down vertically except for the bottom three rows, which, top to bottom, correspond to the agent being sure that the user is in done-state, a specific goal state, and a start state. Red dots indicate that the agent should ask a general question, yellow dots indicate that the agent should make some kind of confirmation, and green dots indicate that the agent should complete some kind of movement. Black dots indicate the robot should do nothing. By scanning across a row, we can see which user beliefs are most discriminative of the underlying preference state.

2. If I don't know where you want to go, should I list all the places I know (instead of asking you where you want to go)?

3. If I'm only 40% sure that you want to go to a certain location, should I still try confirming that location with you?

A side benefit of asking hypothetical questions with the form "If I am... should I...?" is that the query depends only on the user's preference state, not their current belief. Thus, the parameter POMDP remains a POMDP over our original discrete (s_u,$\vec{s_r}$) state space instead of being a POMDP over a continuous ($b(s_u)$,$\vec{s_r}$) state space.

Since the meta-actions are hypothetical, we may be concerned that the agent will ask them during random times and frustrate the user. A small penalty for asking the policy queries ensures that they will be used only when they are relevant. For example, at the beginning of the conversation, the dialog manager may first ask the user, "Where do you want to go?" Suppose the robot then receives a noisy response that is probably "copy machine" but might also be "coffee machine." The robot may follow up with a query to determine how tolerant the user is to mistakes. If the user is fairly tolerant, the robot will continue to the copy machine without further questions; if the user is not tolerant, it may ask a confirmation question next. The branch points in the robot's conversation are likely to make sense to, if not match, branches in the user's conversation (especially since humans are good at placing themselves in others' shoes to rationalize their behavior).

6.1.2 Approach

Table 6.1.2 summarizes our approach. As before, the resulting model is another POMDP. The only change to the POMDP in Chapter 5 is the presence of the policy queries as additional actions. We know that the user's response to a policy query will be determined by their preference state $\vec{s_r}$, however, without doing any computations, we cannot predict what the user's response will be (if we could, we would already know the optimal policy for any user!). To specify the expected response to a policy query, we must solve the new parameter POMDP in two stages.

In the first step, we fix the value of the preference state $\vec{s_r}$. The resulting POMDP only has uncertainty about the user's current intent, and we can solve it with relative ease. For improved

accuracy on policy queries, we seed the sample of user beliefs with the beliefs that we plan to use in our policy queries. For example, if we plan to ask the question, "If I am 90% sure of where you want to go, should I just go?" we would include a belief that places 90% of its probability mass on one state in our belief set. The expected response to a policy query given a preference state $\vec{s_r}$ is the policy of the POMDP induced by fixing the preference state to $\vec{s_r}$.[2]

Once we determine the appropriate responses to the meta-actions as a function of the user's true preference state, we have a fully-specified parameter POMDP. In Chapter 5, we alluded to the difficulties of solving the large parameter POMDP; now the problem becomes even tougher. With explicit rewards, we could assume that space of reachable beliefs would quickly become small since we only had to observe a few rewards to determine our preference state. Thus, we started with a rough solution and refined it as we narrowed in on one part of the belief space. Without explicit rewards to quickly prune the space, our solution must know the correct actions even with large amounts of state uncertainty; in fact, with policy queries, if the choice for the next action is clear regardless of which preference state we are in, the system should simply choose to take that action instead of trying to get more information about its preference state. While beneficial with respect to the dialog—since less feedback is required—we now require a much better solution to our large parameter POMDP. We note that all of the computations can be done off-line, before user interactions begin, but the complexity of solving the large POMDP still limits the number of discrete preference states that we can consider.

6.1.3 Performance

Figure 6-2 shows compares simulation performance with and without meta-actions. Computational constraints limited us to having only four discrete user preference states, but we can see that by asking meta-questions,the dialog manager was able to determine what types of queries and actions would increase the overall reward. In this scenario, the simulated user had fairly harsh penalties—a penalty of -10 for incorrect confirmations and -500 for incorrect movements—corresponding to

[2]Equivalently, we could have found the expected response to each policy query by first solving the parameter POMDP without meta-actions; however, since the matrix computations involved in solving a POMDP scale as O(n^3) in the size of the state space, it is more efficient to solve many smaller POMDPs for each value of $\vec{s_r}$.

Table 6.1: An approach to incorporating policy queries into the parameter POMDP. Rewards are assumed to take on discrete values.

META-ACTION POMDP (DISCRETE REWARDS)

- Solve the POMDP that corresponds to each preference state being fixed.
- Use the policies from those solutions to determine the expected observations as answers to each meta-action.
- Solve the large parameter POMDP (which includes meta-actions).
- Loop:
  - Interact with the user.
  - Wait for an observation.
  - Update belief.

the first user state in Figure 6-1. When no meta-actions were available, the robot had no way of inferring the user's preference state, that is, its belief over the preference state stayed uniform through the entire dialog. While it made more mistakes than the dialog manager that used meta-actions, we see that the dialog manager without any reward feedback still performed reasonably well because it was aware of it uncertainty in the user's true preference state and therefore followed a fairly conservative policy.

Meta-actions not only helped us detect harsh users, but they helped us discover when user is tolerant to mistakes. In Figure 6-3, the user had a -5 penalty for incorrect confirmations and a -100 penalty for incorrect movements (corresponding to preference state four in Figure 6-1). Both policies had similar performance, but in this case, the policy that used meta-actions did slightly better because it realizes that it can actually be less conservative than it initially was. By making quicker decisions, it decreases the total dialog length and avoids asking unnecessary questions.

Finally, we note that as the reward values become more spread, the effects of the meta-actions become more pronounced. For example, if the penalty choices for an incorrect movement are either -500 or -50, then the gap between the learner and the non-learner much larger (see Figure 6-4. In this scenario, the non-learner does poorly because its belief is split between thinking that incorrect movements are as inconsequential as incorrect confirmations and thinking that incorrect move-

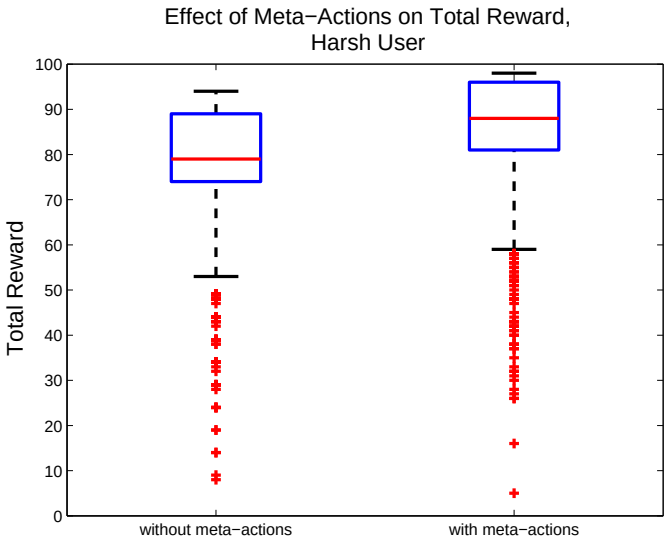

Figure 6-2: Box-plot of total rewards for the scenario where the "real" user has reward -50 for incorrect confirmations and -500 for incorrect movements. By asking about the user's preferences, the dialog manager with meta-actions is able to avoid actions that will frustrate the user. Each simulation had 100 trials of 30 dialogs.

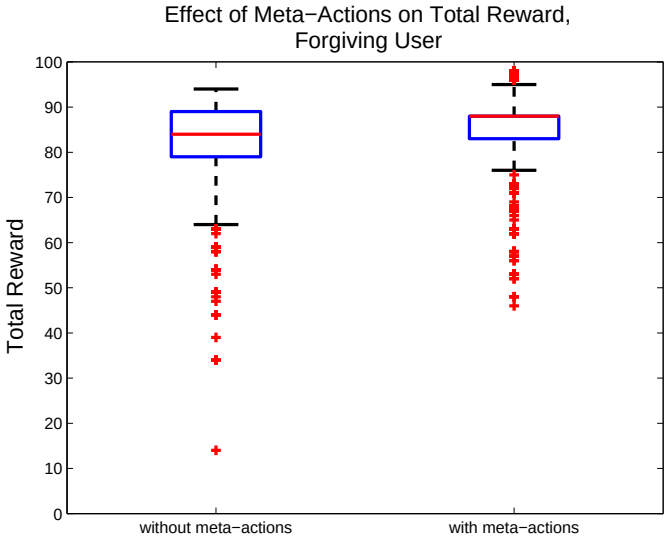

Figure 6-3: Box-plot of total rewards for the scenario where the "real" user has reward -5 for incorrect confirmations and -100 for incorrect movements. The agent still takes meta-actions, but the information is less useful since there reward for an incorrect confirmation, which has a larger impact on the policy, does not contain a major pitfall to avoid. Each test had 100 trials of 30 dialogs.

ments bear a significant penalty. While not very realistic, this example does again demonstrate the utility of meta-actions in resolving aspects of the user's preference state.

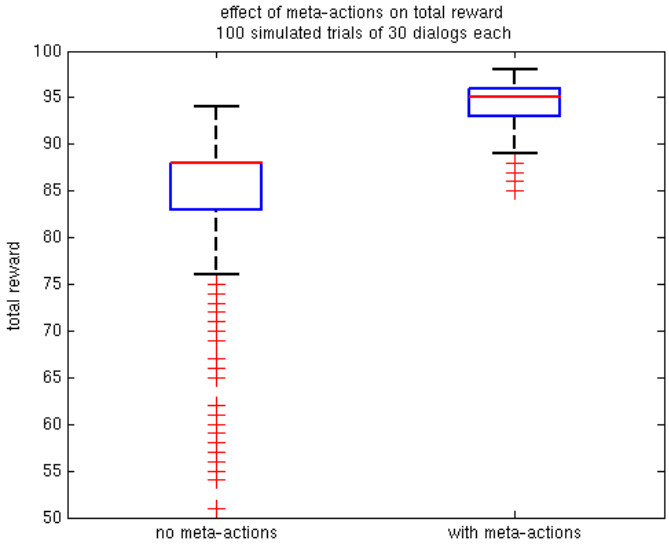

Figure 6-4: Box-plot of total rewards for the scenario where the "real" user has reward -50 for incorrect confirmations and -500 for incorrect movements. In this case, the parameter POMDP had choices -500 and -50 for incorrect movements, and thus the difference in performance is more pronounced than in Figure 6-3. By asking about the user's preferences, the dialog manager with meta-actions is able to avoid actions that will frustrate the user. Each simulation had 100 trials of 30 dialogs.

Next, we tested this initial meta-action model on the actual wheelchair. We provide anecdotal results from a student volunteer talking to the wheelchair in Table 6.2. In the first dialog, the user prefers to be given a list of options rather than open ended questions. We realize that this is somewhat unrealistic scenario, but in some noisy situations, a user may realize that open-ended questions are harder for the robot to interpret than yes-no answers. Once the dialog manager realizes the user's preference, it lists all of five locations that it can drive to around the lab, looping until it is sure that it has heard a confirmation from the user. After the second confirmation, it is sure enough about the user's desire that it does not ask further meta-action queries.

In the second conversation, Table 6.3, the user states that she prefers open-ended queries. Once the user provides the desired location, the robot is fairly certain that the user wants to go to the coffee machine, but does not wish to make a mistake. Thus, it asks the user whether it should confirm when it fairly certain. The user responds to the affirmative, and the robot follows with

Table 6.2: The following dialog (collected from a volunteer in our lab), demonstrates how a typical conversation changes as meta-actions are introduced; in this case the user prefers less open-ended conversations.

With Meta-Actions: Conversation One

This user preferred to be asked confirmation questions.

USER: Hello.

ROBOT: If I don't know where you want to go, should I list options?

The system uses this query to determine if the user is tolerant to incorrect confirmation questions.

USER: Yes.

ROBOT: Do you want to go to Seth's office?

USER: No, I want to go to Nick's office.

ROBOT: Do you want to go to Nick's office?

USER: No, I mean yes.

Unfortunately, the user's response is ambivalent enough that the system does not recognize the user's intent.

ROBOT: Do you want to go to Bryt's desk?

USER: No.

ROBOT: Do you want to go to the group meeting?

USER: No.

ROBOT: Do you want to go to coffee machine?

USER: No.

ROBOT: Do you want to go to Nick's office?

The robot continues to operate by listing locations, but it picks the most likely one to repeat.

USER: Yes.

ROBOT: Going to Nick's office.

another open-ended question (in line with the user's preferences as indicated at the start of the dialog) before executing the action. In completing this dialog, we note that the user was initially surprised by the robot's queries; to be a deployable system the questions should be made easier to understand. Finally, we also note that the two conversations described here are initial conversations with the user—once the robot has determined the user's preference state, it will not ask meta-action queries in further interactions.

Table 6.3: The following dialog (collected from a volunteer in our lab), demonstrates how a typical conversation changes as meta-actions are introduced; in this case the user prefers open-ended dialogs.

With Meta-Actions: Conversation Two
This user preferred to be asked more general questions.
USER: Hello.
ROBOT: If I don't know where you want to go, should I list options?
USER: No, now take me to the coffee machine.
ROBOT: If I'm 95% sure about where you want to go, should I still confirm with you first?
Knowing the user's tolerance to going to the wrong place didn't matter in the previous context, but now it does affect how the conversation proceeds.
USER: Yes.
ROBOT: Where do you want to go?
The robot has some idea of where the user wishes to go from the first utterance, but it needs additional information. It also knows that the user is not tolerant to incorrect confirmations.
USER: The coffee machine.
ROBOT: Going to the coffee machine.

6.2 Discrete Approach: Learning New Words

In this section, we assume that the reward model is known. We also assume there are a discrete set of possible observation models. Our primary goal is not to learn about how noisy the observations are—as we mentioned before, observation noise must be learned through experience, not queries—but to learn the mappings for new words. We have not discussed such a model explicitly before, but the resulting parameter POMDP is similar to that of Section 6.1: instead of states being pairs (s_u,$\vec{s_r}$), states are now (s_u,$\vec{s_o}$) where $\vec{s_o}$ encodes the true mapping for all unmapped observations. Queries about the policy are now replaced by queries about the meanings of new words.

In a relatively static, noise-free environment, we can quickly glean the meaning of a new word from context. For example, if the user first asks to go to the "kiosk" and then follows up by asking for the "information desk," we can infer that the word "kiosk" probably means "information desk" without asking the user a question. In preliminary tests, we found that learning new words simply by listening was generally effective (consider the plots for learning observation models in Section 5.2), and observation queries—which carried a small annoyance penalty—were almost never used (even in cases where repeated general queries were penalized).

In this section, we focus on a particular situation where observation queries can be of use: consider a situation where the user is likely to change their mind about a request. If we do not satisfy the request quickly, the user may change their mind and we may have to discover the user's intent anew. We cannot wait for the user to use a particular keyword. Even worse, if the user changes his or her mind several times in one dialog, it becomes much more difficult to infer the user's true user state at any point in time. Increased uncertainty in the user state makes it tougher to infer the meaning of a new word. Thus, not only can we not afford to wait for the user to use a keyword we already know, but it will take us a long time to learn the meanings of the keywords we should know.[3]

6.2.1 Model Definitions

We continue to use our basic five-goal model as a starting point and set the rewards for general queries, incorrect confirmations, and incorrect movements were -5, -10, and -200 respectively. For our simple example, we add two more unmapped observations to the model. Our discrete observation models consist of fourteen scenarios where each of these observations maps to one or zero goal states. Our new underlying state space consists of the 98 combinations of the seven user states s_u and the fourteen observation states s_o.

Next we defined our observation model. Let p_{ask} be the probability of hearing the correct observation when making a general query (0.7 in our example). Suppose o_1 is the mapped obser-

[3]Although similar, we note that the problem of inferring the underlying user state is not quite the same as the growing state estimation error in Section 4.2.2 because we have finite, discrete set of possible mappings. In theory, our parameter POMDP should still converge to the correct observation state.

vation to state one and the user also uses the unmapped observation $o*$ to refer to state one. In our (somewhat extreme) model, we set $P(o_1|s_1, ask) = P(o*|s_1, ask)/10$ with the constraint that $P(o_1|s_1, ask) + P(o*|s_1, ask) = p_{ask}$. That is, the probability of seeing the unmapped observation was 10 times more likely than seeing the mapped observation, but the total probability of seeing one of the two "correct" observations was still p_{ask}.

We set the p_t, the probability that the user does not change goal state in mid-conversation to 0.75 (instead of the 0.99 in previous experiments). As a result, the mean time for the user to change his mind was only four exchanges instead of 100 before. The system had to act quickly to catch the user before he changed his mind.

6.2.2 Approach

The user's observation state defines the expected answer to the observation query, so we did not require a two stage procedure as we did in Section 6.1. We sampled 7500 initial belief points and used them to solve our POMDP. While iterative resampling may have been beneficial, the initial solution still provided reasonable results. We solved the POMDPs using the approximate backups described in Section 5.2.2; the (straight-forward) procedure is described in Table 6.2.2. We used 100-150 of these backups to ensure that the lookahead was deep enough that the value of knowing the correct observation state became apparent to the agent.

Table 6.4: An approach to incorporating meta-actions into the POMDP. New observations are assumed to take on specific mappings.

META-ACTION POMDP (DISCRETE OBSERVATIONS)

- Solve the large parameter POMDP (which includes meta-actions).
- Loop:
  - Interact with the user.
  - Wait for an observation.
  - Update belief.

6.2.3 Simulation Performance

We tested our toy example in which the dialog manager had the option of asking observation queries of the form, "Which goal (if any) does this word refer to?" In the simulated ground truth, the first unmapped observation was likely to be seen in the first state and the second unmapped observation did not correspond to any of the goal states.

Figure 6-5 shows the mean performance of the policy that used meta-actions compared to a policy that did not. Both policies were trained using the same number of belief samples, based on there respective dynamics, and approximately the same number of backups. The policy with meta-actions approaches the optimal value, and, more strikingly, we see that the policy without meta-actions actually does worse over time. Figure 6-6 sheds some light on why the performance of the system without meta-action queries degrades. At the first interaction, the probability mass on the correct observation state begins at the same value for both systems. The observation queries—which typically occurred in the first to dialogs—helped jump-start the learning process, and over the course of several dialogs, all of the probability mass concentrated in the correct state. Without meta-observation queries, however, the probability mass actually degraded to zero—it is no wonder that the policy does so poorly in Figure 6-5!

In these tests, the probability mass for the dialog manager without meta-actions settled on the observation state where neither unmapped observation has any relevance (not shown). Essentially, the system could not pin down what state the user was in when the unmapped observation was observed and decided that it could rely on the unmapped observations for additional information. In general, we know that even with noisy data, any belief with full support over the observation states should converge to the true observation state. Moreover, the approximate policy chose actions that both failed to illuminate the user's underlying observation state and failed to satisfy the user's goal. Regardless of whether additional sampling or backups would have improved the performance of the dialog manager without meta-actions, the results illustrate that by allowing the system more avenues of gaining information in a noisy and dynamic environment, we reduce the complexity of the solution.

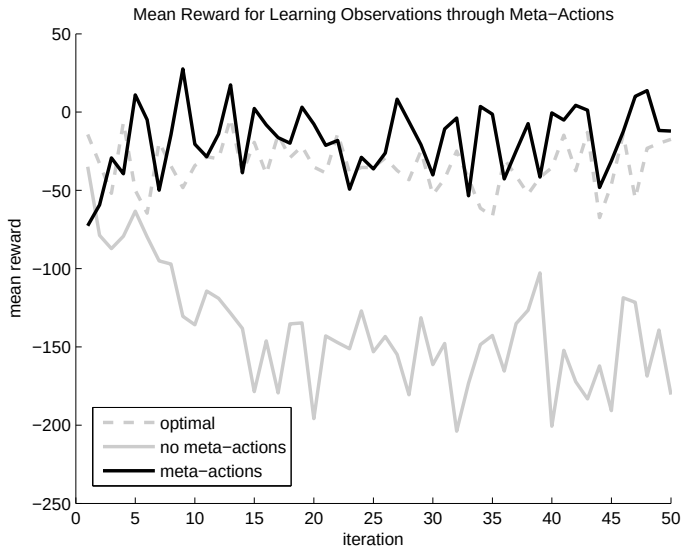

Figure 6-5: Mean performance of a dialog manager with and without observation meta-action queries. The means are aggregated over 200 trials of 50 simulated dialogs.

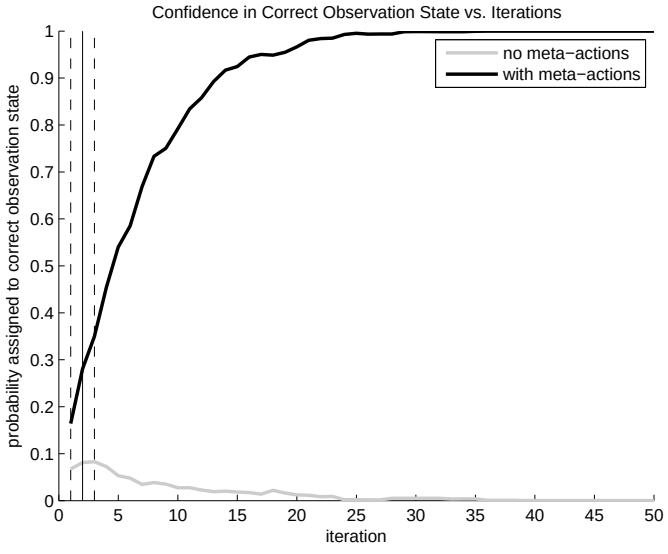

Figure 6-6: Mean probability associated with the correct observation state.

6.3 Continuous Model

Since the meta-actions often require us to have good solutions for large parts of the state space, the approaches in previous two sections, where we assume a discrete set of underlying models, do not scale well to large numbers of models. As in Chapter 5, moving to a continuous model representation trades the ability to (potentially) do deep planning for model richness. We restrict ourselves to case of relatively static environments—that is, the user is not likely to change his mind in mid-dialog—and policy queries. We first demonstrate reward-learning when the observation model is known. In the final part of this section, we consider learning both continuous reward models and continuous (asymmetric) observation models.

6.3.1 Model Definition

As before, we continue to use are basic five-goal model. We assume that the rewards for a correct action (+100) and a correct confirmation (-1) is known, which sets the scaling and translation factor for the reward model (just we did in Chapter 4). We further constrain our reward for incorrect movements to be on [-1000,-20], our reward for incorrect confirmations to be on [-20,-1], and our reward for general queries to be on [-10,-1]. For simplicity, we will use a uniform prior over this three dimensional joint reward space. As in all our models so far, we assume that the symmetric rewards. In the second set of results, when the observation model is also uncertain, we place Dirichlet priors on the observations just as we did in Chapter 4.

Paralleling Section 5.2, we sample a set of POMDPs from the observation and reward priors. At each time step in the dialog, each POMDP updates its belief based on previous observation and action. The actions are chosen by the Bayes risk action selection criterion described in Section 5.2; each POMDP sample contributes to the risk according to its weight. (Initially, each POMDP receives equal weight.) Using a set of POMDPs allows us to get a sense of the spread of policies given the uncertainty in parameters; we reweight and resample them as the priors over the parameters change. Before, we relied only on observations to update the priors over the parameters. Meta-actions will also allow us to learn continuous reward models.

6.3.2 Approach

Without a deep look-ahead, policy queries do not change how we "solve" the POMDP. We solve each sampled POMDP and choose an action based on the Bayes risk. Unlike in Section 6.1, we do not have a fixed set of policy queries determined at the beginning. Instead, we allow the robot to query a policy oracle—that is, the user—at any point when it is confused.[4] Each sampled POMDP will have a different belief based on its observation parameters, but all POMDPs will have a recommended action based on the same sequence of prior actions and observations. If the sampled POMDP recommends an action that differs from the oracle, we know that it is unlikely to be valid.

There are two questions about how these meta-actions should be used. The first is when we should ask a meta-action query. We choose to take a meta-action if the Bayes risk is greater than (that is, more negative than) a certain threshold and if taking a meta-action might reduce that risk. In our tests, we set the threshold to be -1. The number of meta-action queries asked was relatively small, so we did not attempt to tune this hyper-parameter.

The larger question is how to incorporate the information from a policy query. If both the reward space and the observation space are uncertain, then we cannot know if a given POMDP disagreed with the oracle because its reward values were poor, its observation values were poor, or both. There is also a chance that both the reward and the observation models were fine, but our approximate solver handled that particular history poorly. A related issue is if we down-weight all of the POMDPs that suggest an incorrect action, we quickly end up with a sample set where one POMDP has a majority of the weight. The dialog manager is now at the mercy of the best sampled POMDP, whose performance is often much worse than the cautious actions dictated by the Bayes risk criterion.

Since incorporating the results of a policy query is difficult when both the observation and reward models are unknown, we first describe a reasonably efficient approach if only the reward model is unknown. Recall that each sample POMDP is a point in the reward parameter space.

[4] We realize that asking a question of the form "Given our conversation so far, what should I be doing now?" is vague and difficult to answer even for an experienced user, and improving the usability of policy queries is an area for future work.

Figure 6-7 shows two POMDPs (circles) for a two dimensional reward space. Suppose that after a particular history, our user tells us that the correct action is to ask a general question. Presumably, there is some boundary (dashed line) which separates reward instantiations whose policies will agree with the user and those which will not, but we do not know where that boundary lies. However, we do know that if "ask" is the correct action, then not only is the POMDP in the top left corner wrong, any POMDP in the top left hashed region is wrong—decreasing the reward on confirming or increasing the reward on asking will certainly not tip the policy towards asking.

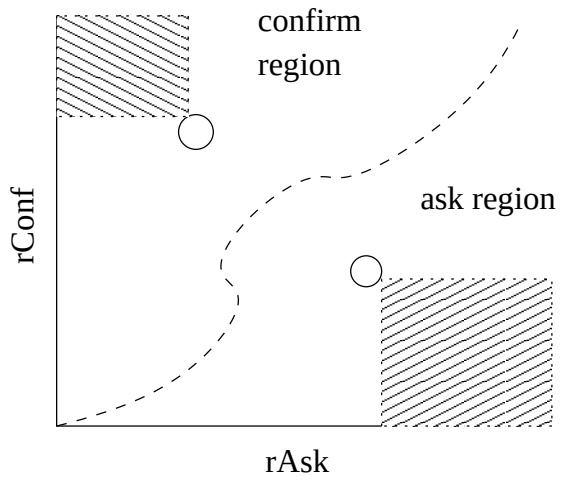

Figure 6-7: This figure shows how policy queries can be used to prune a two-dimensional continuous reward space. Each circle represents a POMDP. POMDPs that suggest incorrect actions can be used to prune or down-weight large sections of the reward space.

The policy of any of the POMDP samples at any point may not be correct, especially since we use so many approximations in our computing our policies. Thus, instead of completely pruning away or removing regions from the reward space, we reduce their likelihood. This is yet another parameter that needs to be tuned: set too high, the system will repeatedly ask meta-action queries just to reduce the weights in some areas. Set too low, valid regions marked as invalid may take a long time to recover. We found that reducing the likelihood of bad regions by 0.1 provided good performance.

As the number of meta-action queries increases, the size of the likely region can become very small, and basic rejection sampling will take a long time to choose a new random instantiation. For more efficient sampling, we re-discretize the reward space into a 3-array. Each block in that array

is assigned a weight. To sample from the array, we sample along one dimension at at time. Using a binary search, this takes $\log(n)$ steps to sample one dimension if there are n partitions along that dimension. For each step, we need to sum over the unassigned dimensions to compute their relative weights. There are probably faster sampling approaches (quad-trees, range trees), but we found this sped up our search enough to make it run in real time. Once we choose a fine enough cell in the reward space, we can sample uniformly from that cell. Thus, if only the reward model is unknown, we can efficiently prune and sample from the reward space.

Unfortunately, when both the reward and observation models were unknown, we cannot reason about the reward space as we did above. The POMDP's action depends on both the observation and reward model, and the high-dimensional observation space cannot even be discretized efficiently to create a joint-space with a reward model for each observation model. Without an efficient representation for the joint observation-reward prior, we are forced to sample from it using rejection sampling. Given a set $\{$(H,a)$\}$ of histories and oracle results from all policy queries to date, we took the following steps to resample a new set of POMDPs:

1. Evaluate how many policy queries that each of the POMDPs in the current sample provided an incorrect action.
2. Sample and solve new POMDPs from the observation prior (updated as we get more observations) and a uniform reward prior. If the new POMDP errs on equal or fewer policy queries than the worst POMDP in the current sample, replace the old POMDP with the new POMDP.
3. Continue sampling until either (1) all POMDPs have error less than or equal to the best POMDP in the original sample or (2) we exceed a certain threshold of samples.

By trying the match the best sample in the current set—instead of satisfying all previous policy queries—the system is tolerant toward variations in approximate solutions.[5] In our actual implementation, we also found it useful to sample a minimum number of POMDPs—even if the POMDPs had the same number of incorrect responses to policy queries, replacing all or most of the POMDPs with equally poor performance in the sample set was helpful since the new POMDPs

[5] We also investigated other stopping conditions, such as trying to only improve the sample set by a fixed proportion of new samples, but these did not perform as well as the approach outlined above.

were more likely to have the correct observation model. Table 6.3.2 summarizes our approach.

Table 6.5: Sampling POMDP approach to solving an uncertain POMDP with meta-actions. The observation and reward parameters are now continuous valued.

META-ACTION POMDP (CONTINUOUS)

- Sample a set of POMDPs from an initial prior.
- Approximately solve that set of POMDPs.
- Loop:
  - Choose the action that minimizes Bayes Risk, or, if risk is great, decide to take a policy query.
  - Wait for an observation.
  - If we did a meta-action, reweight the POMDP samples accordingly. Update the reward prior if we assume that the observation model is known.
  - At the end of a dialog, update observation priors and resample POMDPs.

6.3.3 Simulation Performance

Figure 6-8 shows the simulation performance of meta-actions under the continuous reward model in the case where the observation parameters are known and symmetric. Meta-actions prune the space of the user's preferences and as the POMDPs are resampled and reweighted, we quickly reach close to the optimal level of performance. Since the observations do not provide any information about the rewards, simply reweighting and resampling the POMDPs, as we did in Chapter 5, does not help us learn the user's preferences. Policy queries without resampling also performed poorly because we need to be able to get POMDPs with approximately the correct reward parameters; reweighting without resampling hurts performance because all of the probability mass quickly converges to the least-wrong POMDP in the sample. As a result, we neither have the correct POMDP (which would require resampling), nor do we have a conservative buffer created by several samples (due to the reweighting).

Finally, we tested the approach above against the same simulation model as in the previous

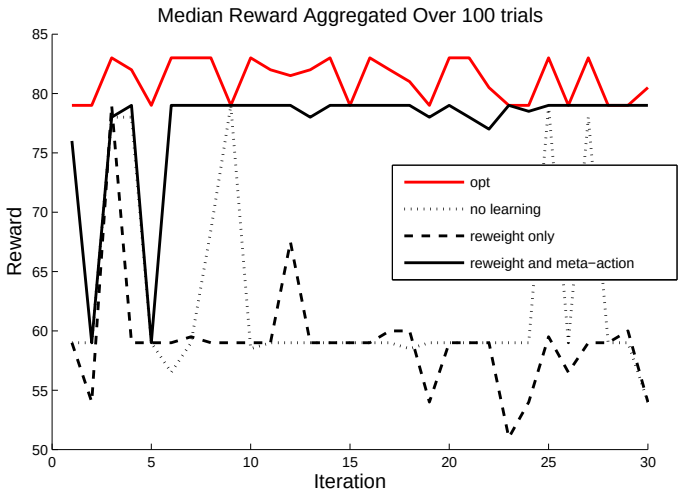

Figure 6-8: Performance of Meta-Actions under the continuous reward model. The observation model is fixed. Without any opportunities to learn about the rewards, reweighting by itself is not useful, but meta-action queries quickly bring us to close to the optimal reward.

chapter when both the observation and the reward model were unknown. Figure 6-9 shows the mean performance of the learning approach. This is the most complex scenario we have consider so far, and as a result, the learning is significantly more noisy. Another factor that makes it difficult to see the effects of the policy queries is that the Bayes risk action-selection criterion, with its conservative choice of actions, also prevents relatively scattered sets of POMDP samples from making poor decisions. We see that the policy-query approach reaches the closer to the optimal faster than simply relying on observations. In Figure 6-10, we see that the policy-query approach also is also less likely to make major mistakes.

6.4 Discussion

We have seen that meta-action queries provide a useful way for the agent to learn about the user's preferences (and, if needed, vocabulary). Especially in the the continuous case, learning from meta-actions is not nearly as clean as the expected value approach in Chapter 4, where the system received clear and explicit reward feedback at every stage. However, we show that we can learn enough of the user model to achieve near-optimal simulation performance without the annoyance of explicit feedback. Asking about actions that we ought to take—along with the conservative

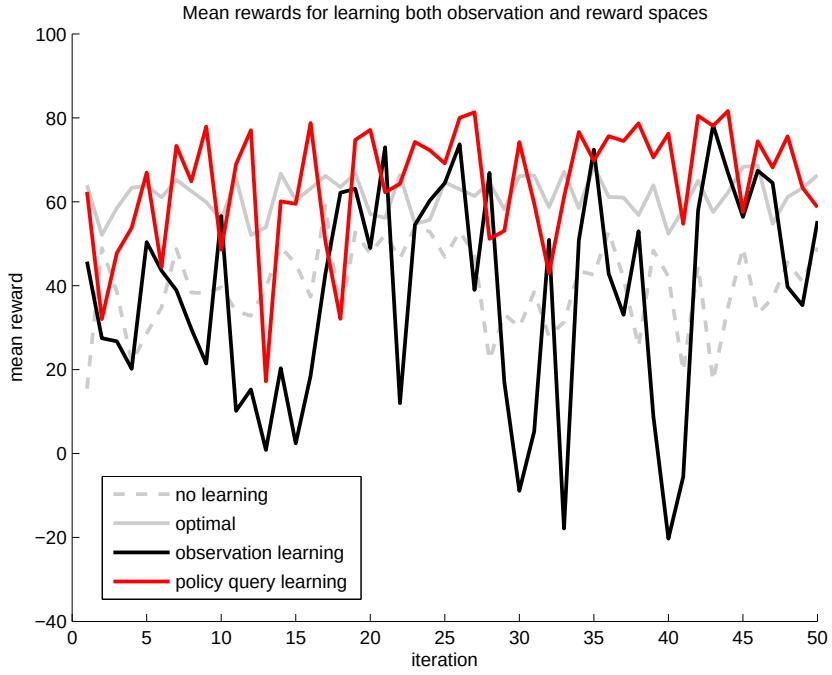

Figure 6-9: Mean performance for learning continuous reward and observation models. The data is noisy, but we can see that the meta-action approach (red) quickly improves upon the nominal performance.

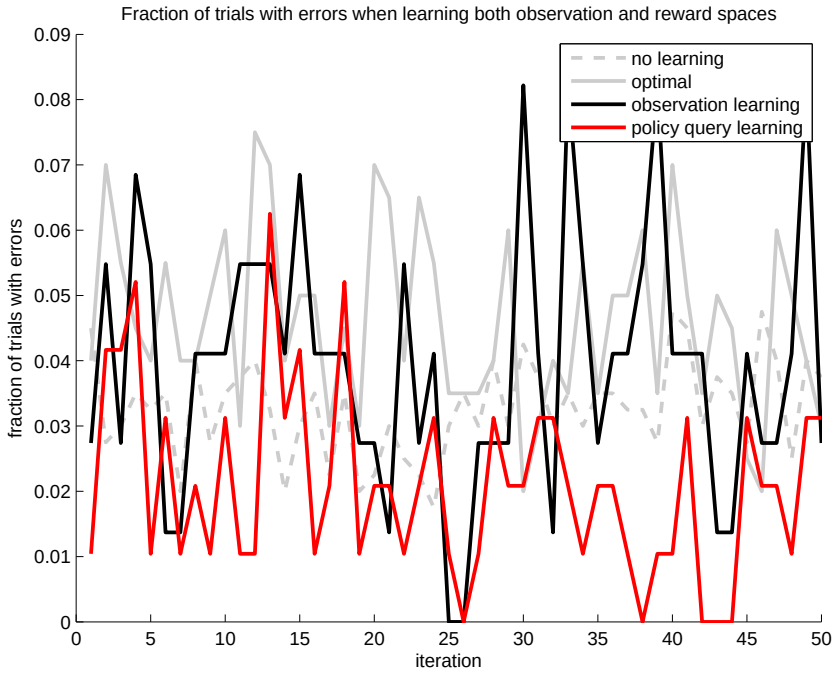

Figure 6-10: Fraction of runs with major errors while learning continuous observation and reward parameters. Again, the policy-learning approach seems to make relatively fewer major errors per run.

Bayes risk action selection criterion—prevents the dialog manager from making serious mistakes.

To improve the meta-action queries, work must still be done to make them more intuitive to the normal user. Even expert users may lack a intuitive grasp of belief states, and they may not provide the answer that matches their true preference model. In the continuous case, we could also benefit from more efficient ways to sample POMDPs from the joint reward-observation prior.

Finally, an even more subtle form of policy learning would use cues such as ‘good work’ or ‘that’s wrong’ from the user to learn both observation and reward models. It is unlikely we could rely solely on such cues because different users may be more or less talkative, but for users who do supply such information, this may be a further approach to reduce the amount of information for which the robot needs to actively prompt the user.

Chapter 7

User Validation on a Complex Model

In this chapter, we present anecdotal observations from a collection of four user tests. Our goal is to show that not only do the policy meta-action approaches developed in Section 6.3 work in the aggregate sense, but they can also help a system adapt to a single user in a single trial. We first present the model and the parameters used to initialize the model. We continue with a description of the system and a qualitative summary of the system's performance.

7.1 Model Definition

The model and the solution approach is identical to that presented in Section 6.3. However, instead of five goal states and eleven observations, we included ten goal states and thirty-eight observations. The first seven goal states were locations to which the wheelchair could drive, while the remaining goal states encoded information goals, such as requests for email, the time, or the weather. Table 7.1 lists all of the goals and keywords used in the experiment. The starred observations were the observations that were initially mapped to the goals. The remaining observations began with no initial mapping.

We included 15 sample POMDPs in our sample set. Our prior assumed that the probability of hearing the expected (starred) observation in that observation's goal state to be 0.6, and the probability of hearing a confirmation correctly to be 0.8. We let our prior be very uncertain, with

Table 7.1: States and Observations for User Test.

States	Nick's office , Bryt's desk, printer, meeting, coffee machine, copy machine, home, email, time, weather
Observations	Nick's*, Bryt's*, print*, meeting*, coffee*, copy*, home*, email*, time*, weather*, done, yup, no, desk, door, office, paper, printer, seminar, group, presentation, water, tea, milk, kitchen, microwave, inbox, message, forecast, machine, window, food, lunch, date, copier, conference, nick, confirm

a pre-observation count of three per state, action pair (that is three observations over 38 possible observations!). Recall that in expected parameter case (Chapter 4), setting the initial confidence to be too low had caused problems because the system quickly became sure of not-yet-converged probability distributions and therefore started to make many mistakes. Keeping a sample set of POMDPs, which indicated a sense of variance in the parameters, and using the Bayes risk action-selection criterion allowed us to learn robustly and quickly with a very uncertain model. We also assumed that the user was very unlikely to change their mind—we initialized the prior with the probability of the user changing his mind to 0.01, and during the actual tests, we assumed that the user was in the same state throughout the dialog.

As before, we began with a uniform reward prior over the penalties for asking a general query, confirming an incorrect goal state, and executing an incorrect action. We did not distinguish between penalties for incorrect movements (which should be more severe) and penalties for incorrect information retrievals (which should be less severe), although that is an aspect we plan to include in future work. The reason for limiting the number of rewards was primarily computational: the higher the state space in which we had to sample, the longer it took us to find samples that we would accept. One change we did make, however, was to increment the penalty for a policy query by -1.5 (starting at -1) after each meta-action. Gradually increasing the meta-action penalty sped up the transition from the system's reliance on policy queries to applying its new-found information and made for less frustrating user experience.

7.2 Experimental Setup

Even with our fast sampling approaches to solving the POMDPs, actions still took approximately 7 seconds to compute the wheelchair's on-board computer and POMDP resampling required several minutes. Therefore, we completed our tests on a fast (2.6 GHz) desktop, where actions took 1 second to compute and POMDP resampling required about 30 seconds. We realize that speed is critical for a real-time system, and making our approach faster is an area we will study in the future.

After a short introduction, which included a summary of the tasks that the wheelchair could provide and a description of the user interface, each user spent between 15 and 20 minutes interacting with the system (about 14-16 dialogs). The user interacted with the system primary through the microphone, and the system responded with through a dialog window on the computer screen (it could also respond with synthesized voice, but we did not have the users wear the audio headset during our tests). Users were encouraged to use different vocabulary when referring to goals and to repeat goals so that the effects of the learning would become apparent in a relatively short set of dialogs, but they were not coached to show any particular kind of preference model when responding to a policy query. All four users (one of which was the author) were graduate students in the lab.

We now describe our simple user interface. Figure 7-1 shows a picture of the initial dialog window. The background color of the window is purple to indicate that the system is currently inactive. The window consists of four parts. First, the large black text indicates the system's dialog to the user. The row of four buttons allows the user to cleanly start, stop, and reset the system if necessary. The task-success button, in conjunction with the radio buttons at the bottom of the window, are used to determine whether the wheelchair should actually execute a motion command. While not strictly necessary, they are a useful safety feature. Finally, the space below the large text (blank) is reserved for feedback regarding the system state.

Figure 7-2 shows the system in an active state—the screen is a bright yellow, to get the user's attention, and a question is presented in the main text. The smaller feedback window indicates that the system is currently busy synthesizing the question as speech. Once a response is received, the window turns cyan (Figure 7-3) to indicate that it is busy processing. The feedback window tells

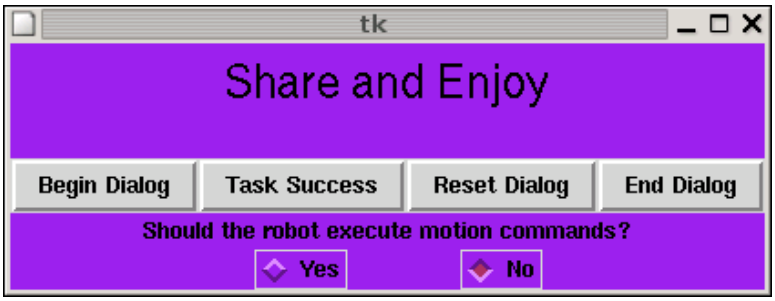

Figure 7-1: Initial dialog window.

the user the system's best estimate for what it heard as well as the fact that it is busy processing. In this particular situation, the system is unsure of what action it should take next—it does not have a strong sense of the user's reward model—and pops up a policy query window (Figure 7-4). Here, the user must use a mouse to choose the most appropriate action. The users were coached to select the action that they would want the system to perform if they were in the system's position.

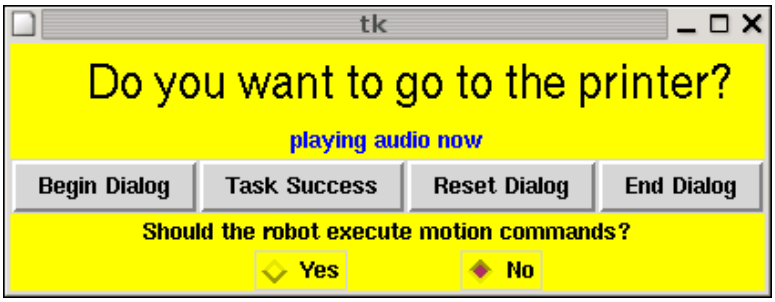

Figure 7-2: Dialog window waiting for user feedback after receiving a speech input.

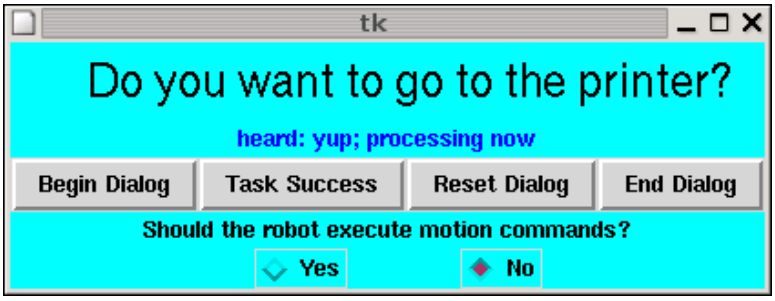

Figure 7-3: Dialog window processing user feedback.

Once an action is chosen, the dialog window again turns bright yellow (Figure 7-5 to indicate that it is taking an action and is ready for user feedback. In this case, the user does indeed wish to

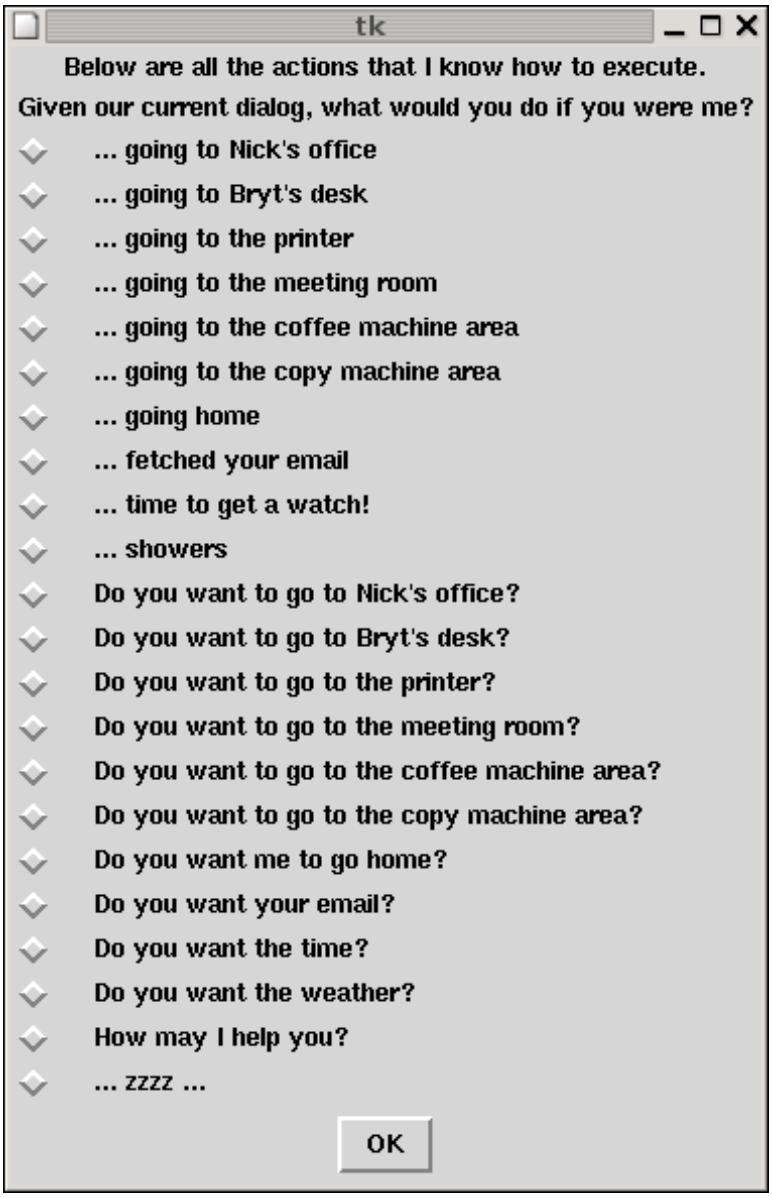

Figure 7-4: The dialog manager is confused and presents policy query window.

go to the printer and clicks the task success button. The window turns inactive (Figure 7-6) as it updates the observation model and resamples POMDPs.

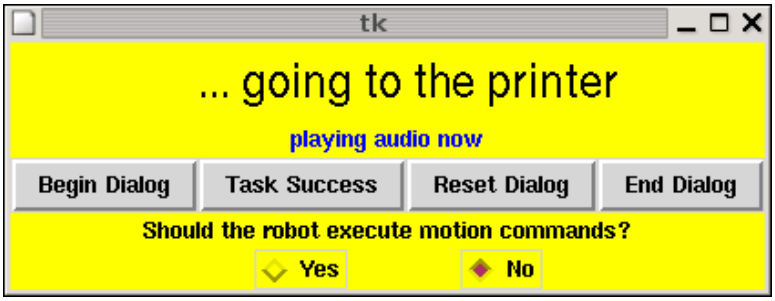

Figure 7-5: The dialog manager completes its task.

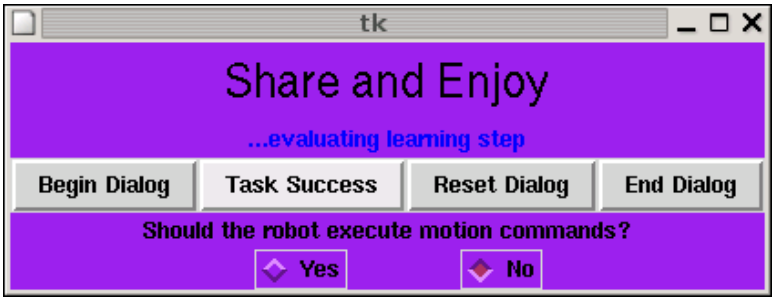

Figure 7-6: The dialog manager retrains its model.

7.3 Results and Discussion

The users were not asked to provide explicit reward feedback to the system, and the goal of our tests were to exhibit the system's basic functionality rather than quantitative improvement along some objective performance metric. Here we summarize the types of adaptation that we observed and highlight the changes through samples from user dialogs. Each of the dialogs is taken from a different user, but the different types of learning that shown here were common across all users. At the end of this section, we include the suggestions we received to improve the user interface.

Learning Basic Dialog. The first pair of example dialogs we present in Table 7.2 are a simple example of policy learning. Because we had set such low confidence levels in our initial obser-

vation model, and we had a very broad reward model, the initial sample of POMDPs contained a large variety of policy options even when the conversation seemed relatively straight-forward. In the first dialog, which occurred early in the conversation, the system is uncertain of what it should do when it hears the word "printer," even though "printer" is a word that already has a specific mapping in its vocabulary and no speech recognition errors occurred. Part of the system's confusion comes from not knowing the reward model—it does not know whether the user would prefer that system asked a confirmation question before taking an action. Another part of the confusion comes from a very imprecise observation model. Since the mean likelihood of hearing the correct observation in a particular state was only 0.6, several models in the initial sample include even smaller probabilities of hearing the correct observation. Thus, the samples are also divided in their belief over the user's intentions. At a later point in the conversation, the system follows the user's recommended course of action when referring to that same state.[1]

Table 7.2: In the following pair of dialogs, the system learns that it should ask a confirmation question after hearing a definitive response to a general query.

Early Conversation
ROBOT: How can I help you?
USER: Take me to the printer.
ROBOT: I'm confused. What action should I take now?
The system pops up the policy query window, and the user clicks on "confirm printer."
ROBOT: Do you want to go to the printer?
USER: Yup.
ROBOT: Going to the printer.
Later Conversation (with the same user)
ROBOT: How can I help you?
USER: Go to the printer.
ROBOT: Do you want to go to the printer?
The system has learned that it should confirm a state before acting.
USER: Yup.
ROBOT: Going to the printer.

Learning New Words. We initially designed the policy queries to be used for learning reward models, but they proved to be quite helpful in learning new words as well. When faced with a

[1]The voice recognition system had trouble with the "s" sound, so we coached the users to use the word "yup" instead of "yes." "Yes" was often confused with "you" or "desk."

new word, the system would often be confused and use a policy query to determine next course of action. Usually, the user would suggest that the system confirm the state that corresponded to their desire. Since confirmation questions were not noisy, a positive confirmation at that stage was generally enough for the system to commit to the correct goal action. Within one or two dialogs, the observation prior would have changed enough so that most of the newly sampled POMDPs would contain the correct mapping for the new word. Thus, with the aid of a meta-action, the dialog could be completed quickly, correctly, and in way that still allowed the system to learn without having to wait until a mapped keyword was heard (compare this initial conversation to the initial conversation in Table 4.7).

Table 7.3 shows an example of such a conversation. The mapped keyword for the weather goal is "weather," but the user is using the word "forecast" instead. In the first dialog, the system uses a policy query to discover what action it should take next, and in doing so, successfully completes the dialog (with the user's help). In the later dialog, it is still not completely sure about the meaning of the new word, but it is able to resolve its uncertainty through a confirmation question instead of a policy query.

Table 7.3: In the following pair of dialogs, the system learns the meaning of a new word through a meta-action query.

Early Conversation
ROBOT: How can I help you?
USER: Give me the forecast.
ROBOT: I'm confused. What action should I take now?
The system pops up the policy query window, and the user clicks on "show weather."
ROBOT: Showers.
Later Conversation (with the same user)
ROBOT: How can I help you?
USER: What's the forecast for today?
ROBOT: Do you want the weather?
The system has partially learned the mapping for the word "forecast," and confirms it before continuing.
USER: Yup.
ROBOT: Showers.

Learning Preferences. In Table 7.3, we saw an example where the user did not want the system to confirm a goal; he wanted the system immediately understand and execute the command. In Table 7.4, we see a set of dialogs where the user always trained the system to respond immediately. As a result, the system gradually moved from asking policy queries to asking confirmations to immediately executing actions. In the second-to-last dialog, we see that sometimes, noise in the observations caused the dialog manager to execute the wrong policy. The system was able to recover using a policy query, but the user was not pleased with this result (especially since observation noise is opaque to the user; he felt as if he had stated his request very clearly). This example shows that while the learning process functioned correctly, the result was not one that the user desired.

Learning Complex Observation Models. Over time, the system was also able to learn much more complex combinations of observation models that accounted for both the user's vocabulary choices and the noise in the voice recognition system. In Table 7.5, we present a situation where the user often referred to the coffee machine area with the word "kitchen" or by asking for "tea." The situation was complicated by the fact that the voice recognition system generally misinterpreted the word "coffee" as "copy," and that the word "machine," (as in "coffee machine" or "copy machine") was also an unmapped word. Initially, the dialog manager is fairly confused, and it takes it twenty exchanges to complete the first dialog[2]. In the later dialogs, however, we see that the system has adapted to the complex observation model. In particular, the second example shows a scenario very much like the early conversation. The system is able to complete the dialog even though new words are used and "coffee" and "copy" are confused with each other. The final example demonstrates that although the system has learned that the word "copy" often occurs in the "coffee" state, it still is able to handle requests for the "copy" state without assuming that the utterance is necessarily a mistake. We note that the system uses confirmation queries to disambiguate the confusion between these two often-confused states.

In general, all users agreed that the system did adapt over time, and the conversations high-

[2]In the interest of full disclosure: these dialogs come from the set completed by the author, and to some extent, initial dialog was purposefully made unforgiving and difficult to confuse the system.

lighted above show how the system was able to learn new words, noise models, and user preferences at once through the use of policy queries. In the seventy total dialogs from the user tests, only two of them had failures where the system actually executed an incorrect movement or information task. While the system was careful, many did express frustration in that sometimes it adapted too slowly. For example, they were generally glad when the system stopped asking policy queries and switched to asking confirmation questions when it was somewhat unsure of the user's intent. However, most of them wanted to know how long it would take the system to forego confirmations and react immediately to their commands.

As the penalty on meta-actions increased, sometimes the system would also take unintuitive courses of action to resolve uncertainty. For example, in one situation, several of the POMDP samples believed that the user wished to go to the printer, but a few confused POMDPs believed that the user wanted the time. The dialog manager chose to confirm whether the user wanted the time—which was not the most likely state in a majority of the POMDPs—before proceeding to the printer. These kinds of actions make sense mathematically, given the reward parameters on those POMDPs, but are unsatisfying to the user.

We also received several suggestions regarding the user interface. Some of these suggestions, such as the color scheme to make the POMDP's internal state more obvious, were incorporated into the user tests. We also adapted the vocabulary set based on what kinds of words the previous user used. One general comment was that the users wanted more information about the system state, and they wanted the dialog manager to seem more intelligent. Some changes were purely aesthetic: even if it was repeating a general query or confirmation, users said that they would prefer that the system rephrased the question to make it seem more dynamic and responsive.

Other suggestions focused on the policy queries. For example, they wanted more detailed system information on the small feedback window. Often they resorted to a side debug window, which included the n-best lists from the voice recognizer, to determine why the system seemed to be misbehaving. While this is may be the first instinct of interested computer science students, their point—that based on the dialog window, it was difficult to infer the system's confusion and determine the correct response to the policy query—is well taken. Along similar lines, users sug-

gested displaying only the top ranked policy options when making a policy query, instead of all of them, to both reduce the number of options and to demonstrate to the user that the system was actually debating between a few reasonable options. We plan to incorporate this feature into future tests.

Table 7.4: In the following set of dialogs, the system learns that the user prefers that the system immediately execute actions without confirmation questions.

Conversation Two: ROBOT: How can I help you? USER: Take me to Bryt's desk. *The system knows the keyword "Bryt" but not the word "desk." In addition, some of the voice recognition system's outputs confuse "me to" with "meeting."* ROBOT: How can I help you? USER: I want to go Bryt. ROBOT: I'm confused. What action should I take now? *The system pops up the policy query window, and the user clicks on "Go to Bryt's desk."* ROBOT: Going to Bryt's desk.
Conversation Ten ROBOT: How can I help you? USER: What time is it? ROBOT: Do you want the time? *The system has stopped asking meta-actions, but it still asks a confirmation before acting.* USER: Yup. ROBOT: Time to get a watch!
Conversation Fourteen ROBOT: How can I help you? USER: Take me to Nick's office. *Most of the speech recognition outputs hear "meeting" instead of "me to Nick"* ROBOT: Going to the meeting room. USER: No, go to Nick's office. *This time, the speech recognition outputs hear a combination of "office" and "meeting" in addition to "no." Hearing both "no" and "meeting," the state it thought the user wanted to go to, makes it confused.* ROBOT: I'm confused. What action should I take now? *The system pops up the policy query window, and the user clicks on "Go to Nick's office." Notice that the user still recommends the system to execute a movement instead of a confirmation.* ROBOT: Going to Nick's office.
Conversation Fifteen ROBOT: How can I help you? USER: Go to the conference room. *We note that conference was a previously unmapped keyword, but the system has begun to learn its mapping by this point in the dialog.* ROBOT: How can I help you? USER: Go to the conference room. *The system has learned not to ask confirmation questions for this user.* ROBOT: Going to the meeting room.

Table 7.5: In the following set of dialogs, the system learns several new words for a particularly noisy pair of states.

Early Conversation ROBOT: How can I help you? USER: I want some tea from the kitchen. ROBOT: How can I help you? USER: The tea is in the kitchen near the coffee machine. *The system sometimes hears "copy" instead of "coffee"; moreover, it's heard so many new words it doesn't know what to do.* ROBOT: How can I help you? USER: Go to the kitchen. ROBOT: I'm confused. What action should I take now? *The system pops up the policy query window, and the user clicks on "Go to the coffee machine area."* ROBOT: Do you want to go to the coffee machine area? USER: Yup. *... The system asks a few more general queries before re-confirming the coffee machine state because it has heard such a varied collection of new words. We note that it does not ask any more meta-actions nor does it prematurely execute an incorrect action.* ROBOT: Do you want to go to the coffee machine area? USER: Yup. ROBOT: Going to the coffee machine area.
Later Conversation ROBOT: How can I help you? USER: Take me to the kitchen. *The system hears a noisy combination of "nick" and "kitchen" (mostly "kitchen").* ROBOT: How can I help you? USER: I want some coffee. *Very unlucky, the system hears* only *the word "copy" instead of "coffee." However, given its noise model and the prior observation "kitchen," it still confirms the correct state.* ROBOT: Do you want to go to the coffee machine area? USER: Yup. ROBOT: Going to the coffee machine area.
Later Conversation ROBOT: How can I help you? USER: I need to make a copy. *As before, the system hears* only *the word "copy." However, without other evidence, it confirms the "copy" state.* ROBOT: Do you want to go to the copy machine area? USER: Yup. ROBOT: Going to the copy machine area.

Chapter 8

Conclusions and Future Work

In this work, we developed and explored a variety of approaches to learn models for dialog management. In the context of controlling robotic wheelchair, we considered the specific scenario where the dialog manager had to determine where to drive the user. With the voice recognition system noisy and the frustrating consequences of driving to the wrong location, we used a POMDP model to trade between information gathering actions (confirmations, general clarifications) and physical movements. Learning the parameters for this POMDP online was the focus of this work.

In Chapter 4, we began with a fast approach that adapted to the user but ignored the uncertainty in the parameters. Since we ignored parameter uncertainty, we could handle relatively complex models—including continuous reward and observation spaces—but the system was sensitive to the initial choice of parameters and sometimes too aggressive. It also required that the user provide explicit reward feedback to train the system.

The remainder of the work built toward handling a continuous observation and action space more robustly. In Chapter 5, we introduced the idea of creating a meta "parameter" POMDP in which the parameters were considered additional hidden state. The large POMDP was difficult to solve, and we used techniques from Chapter 3 to approximate solutions. We also introduced the idea of sampling POMDPs from a prior over the parameters, and using the set of POMDPs to make robust decisions (using a Bayes risk action selection criterion).

Finally, in Chapter 6, we furthered improved robustness and decreased user load using the

concept of meta-actions, or queries about what the system should have done. Meta-action queries allowed us to learn about a user's preferences without depending on the user for feedback. We showed the utility of meta-actions in simulation, and further demonstrated their use in Chapter 7.

All of the approaches in this work used a Bayesian framework to place priors over the parameters and proceeded to solve POMDPs using value-functions. As seen in Chapter 2, value function approximations for POMDPs can be relatively straight forward; however, these techniques force us to either ignore symmetries in the problem (which quickly make the problems intractable) or hand-code the symmetries (which, in addition to requiring expert input, runs the risk of coding incorrect assumptions into the solver). For the ideas presented here to scale to larger systems and more varied problems, future work should consider alternative representations for POMDPs.

One alternative, proto-value function approximation, has been used to find suitable eigenfunctions to approximate the value function in large MDPs[21]. These functions are formed solely from the dynamics of the model, and the underlying assumption is that functions representative of the model dynamics will also be representative of the value function. Whether that assumption is true in the dialog management domain is unclear, but as we have seen already in our current work, some approach to taking advantage of the state and action symmetry in our problem will be crucial to making any value-function based approach efficient and tractable.

Another alternative we would like to consider in our future work is to plan in the policy space, eliminating value functions completely. For example, we would maintain a particle-filter of policy trees or finite-state controllers. We note that despite the large number of parameters we had to learn in this work and the computational difficulties involved, the problems what we considered really had very simple policies. Moreover, in Chapter 6, we encountered significant difficulties in trying to sample from a complicated parameter space just to find a set of parameters that were consistent with information provided from policy queries. Especially for dialog management problems—where polices are simple and policy advice is easy to give—it seems that the policy space may be the right space to view these problems.

In particular, since small POMDPs are quick to solve, it may be possible to construct a policy-tree online that considers only how meta-actions may help the dialog manager branch to more

certain parts of the belief space (this is approach would be similar to the various online policy tree approaches inspired by [23]). On the other hand, while it is difficult to map the effects of meta-actions to sections of the belief space, a finite-state controller approach (based on the ideas of [17] or [2]) may have more direct interpretations in the policy space.

Finally, we plan to conduct more sophisticated user studies with the robotic wheelchair. By measuring how long it takes the system to fulfill a user's needs and surveying users on their satisfaction, we hope to determine if there exists a significant difference between a hand-crafted dialog manager and the kinds of learning POMDP dialog managers considered in this work. The difficulties that users have operating the wheelchair with either model may provide additional insight into what dialog management research questions would add the most value for wheelchair users.

Appendix A

Hardware

Here we provide a brief description of the robotic wheelchair that was designed and built as the target application of this work (see Figure A-1). Our goal is to provide the reader with a general sense our hardware configuration and user interface; a follow-on technical report will contain detailed information for those who are interested in replicating our hardware.

Figure A-1: A robotic wheelchair was the target application of this work.

Modifying a commercial chair seemed to be the right cost-work balance between buying a pre-roboticized wheelchair and machining our own wheelchair from scratch. We chose a Jazzy 1103 power chair from Pride Mobility because of its center-wheel drive (for ease of control) and reasonable cost. The Jazzy 1103 also had an adjustable seat that could be raised to create room for custom parts and wiring, including the on-board computer and power bus. To control the wheelchair, we intercepted the joystick signals and replaced them with our own translation and rotation commands. While somewhat indirect, this approach allowed us to command the wheelchair using low power signals as well as create a simple override circuit that ensured that the user would be able to take control if the computer failed.

Used for both obstacle avoidance and localization, the wheelchair's primary mode of sensing is through the laser range finder mounted in the front of the wheelchair. We found that the one scanner was effective for most tasks; however, given the height of the wheelchair, the user occasionally had to provide their own obstacle avoidance for taller protrusions (such as tables) that were invisible to the scanner. Shaft encoders mounted to the motor shafts of both drive wheels supplemented the laser range measurements with odometry information.

The Carmen Robotics Toolkit [22] (with some modifications for the new hardware set-up) provided basic localization, navigation, and mapping capabilities. The dialog manager itself was implemented in Python and Matlab, and interfaced with Carmen via the pyCarmen. All computation was completed on-board except for the voice recognition, which occurred remotely with a system developed by the MIT Spoken Language Systems Group [9]. (The voice recognition was computed off-board for convenience: as research software, the voice recognition system was not yet an easy to install package. There was no computational barrier to computing the voice recognition on board the wheelchair.)

Several steps needed to be taken before any user interactions occurred. First, we drove the wheelchair around an area to collect laser data and build a map. Next, the map was hand-labelled with locations of interest, such as elevators or offices. (In the future, we hope to make it possible to easily add new locations online). The interface to the dialog manager received a list of keywords that corresponded the each location or task (the mapped keywords) as well as a larger set of

keywords to listen for (the unmapped keywords).

The user interacted with the system primarily with a microphone. Once in listening mode, the system continuously monitored the microphone for sustained inputs over a pre-specified power threshold. This simple approach to detecting and end-pointing speech inputs proved to be sufficient for indoor office environments, although more sophisticated approaches may be needed for noisier situations. Once the voice recognition system processed the speech input, the dialog manager scanned the returned n-best list for occurrences of keywords. We did not make any attempts at natural language understanding. At the end of a dialog, the user confirmed the system's final action with a mouse-click. While not strictly necessary, the additional confirmation step prevented the wheelchair from moving prematurely during tests; we wished to avoid safety issues where the user might be trying to talk to the wheelchair and monitor its movement at the same time.

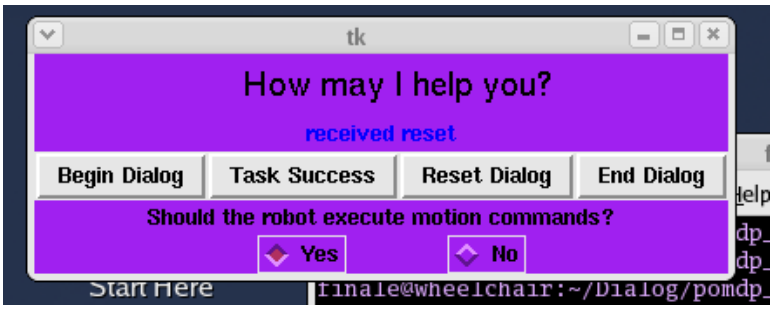

Figure A-2: The user interface. This screen shot shows the window in "quiet" mode, just after the system has been reset. The buttons allow the user to manually reset the system as well as regulate the execution of movement commands.

Finally, a window displayed on a monitor attached to one of the arms was the main form of feedback (see Figure A-2 for a screen-shot). We kept the window small to allow for easy viewing of other navigation related windows as well as leaving the user the opportunity to run programs if desired. The larger window text displayed the system's queries, while the smaller text displayed information about the system's internal state (e.g. "I'm thinking..."). To show attention, the window also changed color—from purple to yellow—whenever the microphone power went over the pre-specified threshold. For experiments where explicit reward feedback was required, the window also included several buttons for each reward input.

Bibliography

[1] Anthony Cassandra, Michael L. Littman, and Nevin L. Zhang. Incremental Pruning: A simple, fast, exact method for partially observable Markov decision processes. In Dan Geiger and Prakash Pundalik Shenoy, editors, *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–97)*, pages 54–61, San Francisco, CA, 1997. Morgan Kaufmann Publishers.

[2] L. Charlin, P.Poupart, and R.Shioda. Automated hierarchy discovery for planning in partially observable environments. In B. Schölkopf, J.C. Platt, and T. Hofmann, editors, *To appear in Advances in Neural Information Processing Systems 19*, Cambridge, MA, 2007. MIT Press.

[3] Mercan Karahan Computer. Combining classifiers for spoken language understanding.

[4] Richard Dearden, Nir Friedman, and David Andre. Model based bayesian exploration. pages 150–159, 1999.

[5] Finale Doshi and Nicholas Roy. Efficient model learning for dialog management. In *Proceedings of Human-Robot Interaction (HRI 2007)*, Washington, DC, March 2007.

[6] Finale Doshi and Nicholas Roy. Efficient model learning for dialog management. In *Technical Report SS-07-07*, Palo Alto, CA, March 2007. AAAI Press.

[7] Eyal Even-Dar, Sham M. Kakade, and Yishay Mansour. Reinforcement learning in pomdps without resets. In *IJCAI*, pages 690–695, 2005.

[8] A. Fern, S. Natarajan, K. Judah, and P. Tedepalli. A decision-theoretic model of assistance. *IJCAI*, 2007.

[9] J. Glass. A probabilistic framework for segment-based speech recognition. *Computer Speech and Language*, (17):137–152, 2003.

[10] Geoffrey J. Gordon. Stable function approximation in dynamic programming. In *Proceedings of the Twelfth International Conference on Machine Learning*, San Francisco, CA, 1995. Morgan Kaufmann.

[11] Eric A. Hansen. An improved policy iteration algorithm for partially observable MDPs. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[12] J. Hoey and P. Poupart. Solving pomdps with continuous or large discrete observation spaces, 2005.

[13] J. Hoey, P. Poupart, C. Boutilier, and A. Mihailidis. Pomdp models for assistive technology, 2005.

[14] Michael R. James, Satinder Singh, and Michael Littman. Planning with predictive state representations. 2004.

[15] Robin Jaulmes, Joelle Pineau, and Doina Precup. Learning in non-stationary partially observable markov decision processes. Workshop on Non-Stationarity in Reinforcement Learning at the ECML, 2005.

[16] M. Kearns, Y. Mansour, and A. Ng. Approximate planning in large pomdps via reusable trajectories, 1999.

[17] Kee-Eung Kim, Thomas Dean, and Nicolas Meuleau. Approximate solutions to factored markov decision processes via greedy search in the space of finite state controllers. In *Artificial Intelligence Planning Systems*, pages 323–330, 2000.

[18] Diane Litman, Satinder Singh, Michael Kearns, and Marilyn Walker. NJFun: a reinforcement learning spoken dialogue system. In *Proceedings of the ANLP/NAACL 2000 Workshop on Conversational Systems*, Seattle, 2000.

[19] Diane J. Litman and Shimei Pan. Designing and evaluating an adaptive spoken dialogue system. *User Modeling and User-Adapted Interaction*, 12(2-3):111–137, 2002.

[20] Michael L. Littman, Anthony R. Cassandra, and Leslie Pack Kaelbling. Learning policies for partially observable environments: Scaling up. In Armand Prieditis and Stuart Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 362–370, San Francisco, CA, USA, 1995. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

[21] Sridhar Mahadevan and Mauro Maggioni. Value function approximation with diffusion wavelets and laplacian eigenfunctions. In *NIPS*, 2005.

[22] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*, volume 3, pages 2436–2441, Las Vegas, NV, October 2003.

[23] Andrew Y. Ng and Michael Jordan. PEGASUS:A policy search method for large MDPs and POMDPs. pages 406–415.

[24] A. Nilim and L. Ghaoui. Robustness in markov decision problems with uncertain transition matrices, 2004.

[25] Tim Paek and Eric Horvitz. Optimizing automated call routing by integrating spoken dialog models with queuing models. In *HLT-NAACL*, pages 41–48, 2004.

[26] J. Pineau, G. Gordon, and S. Thrun. Point-based value iteration: An anytime algorithm for pomdps, 2003.

[27] J. Pineau, G. Gordon, and S. Thrun. Point-based approximations for fast pomdp solving. (SOCS-TR-2005.4), 2005.

[28] Joelle Pineau, Nicholas Roy, and Sebastian Thrun. A hierarchical approach to pomdp planning and execution. In *Workshop on Hierarchy and Memory in Reinforcement Learning (ICML)*, June 2001.

[29] Josep M. Porta, Nikos Vlassis, Matthijs Spaan, and Pascal Poupart. An point-based value iteration for continuous pomdp. 2006.

[30] Pascal Poupart, Nikos Vlassis, Jesse Hoey, and Kevin Regan. An analytic solution to discrete bayesian reinforcement learning. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 697–704, New York, NY, USA, 2006. ACM Press.

[31] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[32] M. Ravishankar. *Efficient Algorithms for Speech Recognition*. PhD thesis, Carnegie Mellon, 1996.

[33] N. Roy, J. Pineau, and S. Thrun. Spoken dialogue management using probabilistic reasoning. In *Proceedings of the 38th Annual Meeting of the ACL*, Hong Kong, 2000.

[34] Stephanie Seneff and Joseph Polifroni. Dialogue management in the mercury flight reservation system. In *ANLP/NAACL 2000 Workshop on Conversational systems*, pages 11–16, Morristown, NJ, USA, 2000. Association for Computational Linguistics.

[35] Trey Smith and Reid Simmons. Heuristic search value iteration for pomdps. In *Proc. of UAI 2004*, Banff, Alberta, 2004.

[36] E. J. Sondik. *The Optimial Control of Partially Observable Markov Processes*. PhD thesis, Stanford University, 1971.

[37] Matthijs T. J. Spaan and Nikos Vlassis. Perseus: Randomized point-based value iteration for POMDPs. *Journal of Artificial Intelligence Research*, 24:195–220, 2005.

[38] Christopher Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.

[39] J. Williams and S. Young. Scaling up pomdps for dialogue management: The ”summary pomdp” method. In *Proceedings of the IEEE ASRU Workshop*, 2005.

[40] Jason D. Williams, Pascal Poupart, and Steve Young. Partially observable markov decision processes with continuous observations for dialogue management. In *Proceedings of SIGdial Workshop on Discourse and Dialogue 2005*, 2005.

[41] Jason D. Williams and Steve Young. Partially observable markov decision processes for spoken dialog systems. *Comput. Speech Lang.*, 21(2):393–422, 2007.

[42] Huan Xu and Shie Mannor. The robustness-performance tradeoff in markov decision processes. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*. MIT Press, Cambridge, MA, 2007.