Optimal Policies for Partially Observable Markov Decision Processes

Anthony R. Cassandra

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-94-14
August 1994

Optimal Policies for Partially Observable Markov Decision Processes

Anthony R. Cassandra*
Department of Computer Science
Brown University
Providence, RI 02912

February 27, 1995

1 Introduction

Decision making is an important task in everyday life. Making the correct decision in some circumstances can help utilize resources better or even be the difference between life and death. There are many applications in which automated decision making is desirable (automated stock trading) and others in which it is necessary (e.g., real time flight controls of a supersonic jet). Unfortunately, at this time, the process of human decision making procedures for many types of problems is difficult or impossible to formalize. Given this limitation, we would still like to have automated decision making perform as well as possible.

An often used and sometimes convenient way to make a machine aware of the dynamics of the domain in which it will be making decisions is to provide the machine with a model of the world. For some domains this is quite acceptable, since the dynamics of the world may be well defined. For other domains, it may be impossible to specify the dynamics fully. Even for

*This work was done jointly with Leslie P. Kaelbling and Michael L. Littman. Anthony Cassandra's work was supported in part by National Science Foundation Award IRI-9257592.

algorithms that do not rely upon being given a model [18], there is a structure imposed upon the environment which acts as a somewhat more abstract model. In this paper, we will be exploring a specific model-based scheme in which decisions need to be made. Even when we cannot assume we have the model, we can use techniques, [2], that allow us to approximate the model and then apply these model-based schemes. The many problems associated with such models will be outlined in a subsequent section. Throughout this discussion, the term *agent* will refer to the automated process that has to make decisions. A convenient example of an agent is that of an autonomous robot trying to survive in a real world environment. However, the agent can simply be a computer program such as one that does medical diagnosis. In this case, the model of the world might be based upon statistics and medical research.

The model we will explore assumes that the agent exists in a unique state at any time point. Furthermore, there are only a finite number of different possible states it can ever be in. For the robot example, these states can be different locations in the world or even the agent's state of knowledge at a given time. Time is assumed to pass in discrete increments (ticks) of some clock and the agent must choose some action to perform at each tick of the clock (it could be to do nothing). Each time the agent performs an action, it will move to a new state, though it is possible for it to remain in the same state. An alternative, but equivalent, view of the agent and world is one where the world exists in some state and the agent interacts with it.

If a machine (computer) is going to make decisions by itself, it needs to have some metric that it can use to differentiate between the good and bad choices. This is another potential trouble spot, because it is often difficult for people to articulate exactly what the differences are between good and bad choices. Nevertheless, we must do so if we desire to have the machine choose between a number of options. The choice in our model is to assume that each state has an associated reward for performing each possible action in that state. This reward reflects the immediate desirability of performing a particular action given that we are currently in some state.

These rewards do not necessarily have to be something the agent actually gets as it operates. We define these rewards as a way of assigning relative values to different states of the environment. Given these abstract values, the agent can then attempt to get to the locations that it knows have higher value. Although we can view the rewards as abstract entities, they might

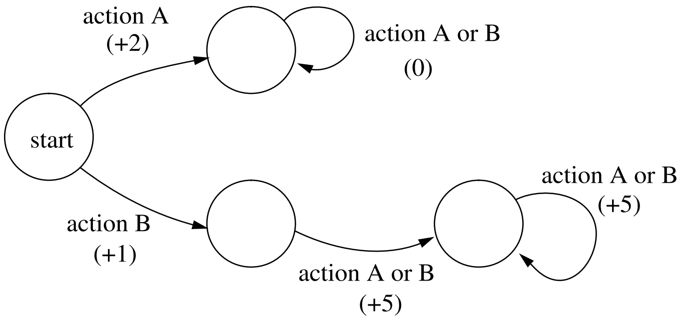

Figure 1: Sample environment showing why one step greedy strategy is undesirable

well be grounded in some real phenomenon that the agent can perceive.

The model described thus far has dynamics as follows: The agent starts in a state i at time t, performs action a, receives an immediate reward for that action in that state and then finds itself in state j at time $t+1$. The dynamics of an environment specified with such a model can be described with just a couple of tables. One table specifies the next state based upon the current state and the action chosen. The other specifies immediate rewards for all combinations of states and actions.

Although this model is much simpler than the one that is actually treated in this paper, it is useful to think about how an agent would use this information to make decisions at each time step. One fairly intuitive notion is that the agent wants to get as much reward as possible after all, these rewards are our way of informing the agent what state-action pairs we think are desirable. But exactly what is meant by "as much reward as possible?" Do we simply want to perform the action that gets us the most reward for this particular step? The answer is probably not. Figure 1 is an environment in which we can get a reward of 2 now, but no more rewards afterwards no matter what actions we decide to take. Also in this environment there is another action we can perform which will only give us a reward of 1 now, but after we take this action we will be able to get a reward of 5 for the next action (and even more with subsequent actions).

The above situation shows an example in which we would probably want to take into account the rewards we might receive in the future, not just the immediate rewards. If we knew in advance how many time steps (call this number k) we had to make decisions for, then we could simply look at the model and decide which sequence of k actions (from a given starting state) will give us the maximum reward. Of course, there could be many possible sequences to consider, but there would always be a finite number of them. However, what if you don't know how many decisions the agent will have to make? In this case we need to consider the future rewards in some other manner. Dealing with this is a somewhat controversial topic. For this paper we will consider an infinite horizon with a discounting factor. The justifications for using this are given later, as are the drawbacks of discounting. Here we will just present the basic concept behind this approach.

The term *infinite horizon* is used to indicate that we will consider rewards for an infinite number of time steps. This is why we need a discount factor. If we merely try to sum rewards forever, we will get an ever increasing sum and all actions will look as if we can get an infinite amount of reward. By adding a discount factor we can cause these infinite sums to converge to a number which we can then use in a comparison to decide which actions are best.

Although we wouldn't expect the agent to make decisions forever, the infinite horizon with discounting is a good model for the instances in which we do not know for how long the agent will have to make decisions. If we assume that the agent will stop making decisions at each step with some probability, then it makes sense not to count rewards that could be received far into the future as much as rewards that could be received closer to the present. The discount factor does this by discounting rewards based upon how far in the future they could occur.

We now come to the point where we would like to make our model more robust. In the previous model description we assumed that each action will always have the same output given a specific starting state. Many domains of interest, however, do not possess this property. More specifically, an agent that is a mechanical device is subject to physical limitations and tolerances. Trying to move a robot 10 inches by applying the exact amount of voltage for exactly the proper time is a difficult thing to accomplish and even if this is overcome, there may be slippage in the wheels depending upon the floor surface, humidity, dirt, etc. So if you've modeled an environment in which one of the actions is for a robot to move 10 inches and another is to move 9 inches, the action to move 10 inches might not succeed. Although the results of the robot's actions are in a sense deterministic, the amount of knowledge

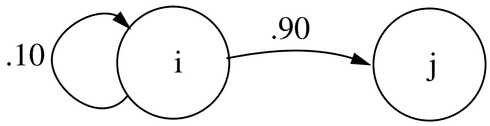

Figure 2: Sample environment showing stochastic actions

that we would have to have to predict it is overwhelming (e.g., where each dirt particle was, coefficient of friction of carpet for the current atmospheric conditions, etc.) Instead of getting bogged down in all these details, the standard approach is to model the actions probabilistically. Instead of saying that performing action a in state i will result in our next state being j, we say that 90% of the time this will happen, but that 10% of the time we could remain in state i. This is shown pictorially in Figure 2.

You can add these action probabilities into the model and get a more robust model. The nice part about this is that you can still figure out what the best action to take is. It is a little more complex than models without probabilistic actions, but solution techniques exist. The technical name for this type of model is a Markov Decision Process (MDP) and it is a very thoroughly explored model in mathematics, operations research, computer science, and related fields.

There is one extremely important element that we have assumed up to this point, but which is not always a good assumption. When we are choosing the best action to take, we first need to know what state we are in. How do we know what state we are in? Before we had probabilities associated with the actions, we always knew the next state after we took an action, so as long as we knew where we started we could tell exactly where we were. Now that we have have introduced uncertainty in the outcome of actions, how do we know what state we actually did end up in? The easy answer is the one that is assumed in the previous model: Just look. The problem is that it is not always possible or realistic to assume you can tell exactly what state you are in at a given time. For instance, the robot example demonstrated the physical limitations of machines. These limitations apply not only to the actions they can perform, but also to the things they can observe in the world. If a robot is 18 inches from a wall and decides to move 10 inches, how will it know whether it is in the state of being 8 inches or 9 inches from the wall? Even if the robot has some sensors that can very accurately tell the distance to the wall, there is still some probability that they could malfunction. The typical case is that mobile robots have crude sensors which are nowhere near 100% reliable. Even if there we perfect or near perfect sensors, the agent will rarely be able to see the whole world at once.

Dealing with uncertainty in observation is the main issue which this paper explores. The previously described MDP model can be made more general by introducing uncertainty in observations. This type of model is referred to as a partially observable markov decision process (POMDP) and is the focus of this work. Many existing techniques utilize the MDP model [7, 3] and the assumption of complete (reliable) observability is one of the major drawbacks of these techniques and those that build upon these. This paper explores existing techniques for POMDPs, presents a new algorithm and shows the usefulness and limitations of the POMDP model in general.

2 The Model

2.1 The MDP Model

A Markov decision process, MDP, is a fairly simple model with a fancy name. It consists of a finite set of states, a finite set of actions and a reward structure defined for each state-action pair. For a robot navigation problem, the states can be viewed as the location of the robot in the environment. Though most environments are really a continuum of states, it is often convenient to discretize these continuous spaces. The actions are the things the agent can do, such as move forward, turn left, pick up an object, etc. Associated with performing each action is an immediate reward. Because the effectiveness of performing an action can depend upon what state the agent is in, the MDP model actually assigns an immediate reward for each combination of states and actions. For example, if the robot is trying to get to a specific location, then performing the action *move forward* when it is next to and facing that location should result in a high reward. If the robot was directly in front of a descending stairway (and it did not know how to walk down stairs), then we would expect the action *move forward* to give much less reward.

Formally, an MDP is a quadruple, $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{W} >$. $\mathcal{S}$ is a finite set of states with state i denoted by s_i. $\mathcal{A}$ is a finite set of actions with action i denoted by a_i or sometimes just a when we refer to a specific action. $\mathcal{W}$ is the reward structure, which is slightly more general than the one described previously. Here, w_{ij}^a represents the immediate reward the agent will get if it performs action a while in state s_i and moves to state s_j. Although this model is more general, we can simplify the reward structure to the one described previously by taking a weighted sum over all possible next states. The weight used in this sum is the probability that we actually move to each of the next states.

$$q_i^a = \sum_j p_{ij}^a w_{ij}^a \tag{1}$$

Alternatively, we could just describe the q_i^a rewards directly. The w_{ij}^a are the actual rewards that would be received, but the q_i^a are the *expected* rewards representing what we would expect to receive on average in the long term.

2.2 Policies

A *policy* is a function or mapping that tells the agent what action to execute. It typically depends upon the state that the agent is in. For instance, this current state could be its physical location. A policy completely specifies the appropriate action for each possible situation (state) that occurs in the model. A policy should not be confused with a plan. A plan is a sequence of actions to perform, and does not necessarily specify the appropriate action for each possible situation. In some sense, a policy is the most general possible plan, since it completely specifies the action to take for each possible state of the environment.

A policy can be either *deterministic* or *stochastic*. A deterministic policy is one that specifies a single action to take in each state. A stochastic policy specifies a number of possible actions to execute in each state.. In addition to a set of possible actions, the agent is given a probability distribution over the set of actions. The agent will then stochastically choose one action according to those probabilities. For this paper will we only concern ourselves with deterministic policies. Work in stochastic policies include [15]. Note that deterministic policies are really a subset of stochastic policies where the probability distribution assigns one action probability 1.

Policies (both deterministic and stochastic) can also be categorized as *stationary* or *non-stationary*. A stationary policy is independent of time; the same policy is applied regardless of when the policy is used. Thus, the policy only depends on the state of the agent and/or environment. A non-stationary policy is dependent on time; the time the policy is to be used will affect what action will be taken. In this case, the time and state and/or environment determine what action the agent should execute.

Policies can also be grouped by the amount of memory required. As an agent moves about its environment performing actions, it is building up a history of its movements. In some situations it might be desirable to remember this history and perform future actions based upon this. A *memory-less* policy is one that uses no history at all. The choice of action depends only on an immediate situation of the agent. A k-memory policy chooses actions based upon the last k pieces of the agent's history.

There are also policies that require a finite amount of state, but the amount of state doesn't directly correspond to the last k pieces of the agent's history. For example, a policy that depends upon whether a button was ever pressed, requires only a single bit of information, but this single bit does not correspond to k-memory for any fixed k.

In this paper we will focus on deterministic stationary policies with memory. We will denote the entire space of these deterministic stationary policies as Δ where a specific deterministic stationary policy is denoted by δ. We will denote a deterministic non-stationary policy as a set of stationary policies, $\delta = \{\delta_0, \delta_1, \ldots, \delta_t\}$ where $\delta_i \in \Delta$ is the stationary policy to use when there are i steps remaining.

2.3 Infinite vs. Finite horizon

In this paper we will discuss two ways to construct optimal policies for POMDP models. Both of these also apply to MDP models and, again for simplicity, it is in the MDP framework that we first present them.

We first consider problems in which the agent only has to make a known finite number of decisions. We refer to this class of problems as *finite horizon* problems; a *k-horizon* problem is one in which the agent will make decisions for k time steps. This period of k time steps is usually referred to as the agent's *lifetime* or the size of the horizon. The actual structure of a finite horizon policy consists of a sequence of deterministic policies, one for each time step (i.e., a determinisitic non-stationary policy). The policy, δ_i, is a complete mapping from all states to actions, $\delta_i : \mathcal{S} \rightarrow \mathcal{A}$. The policy takes the current state of the agent as an argument, so the action specified by the policy for state s_i is denoted by $\delta_i(s_i)$.

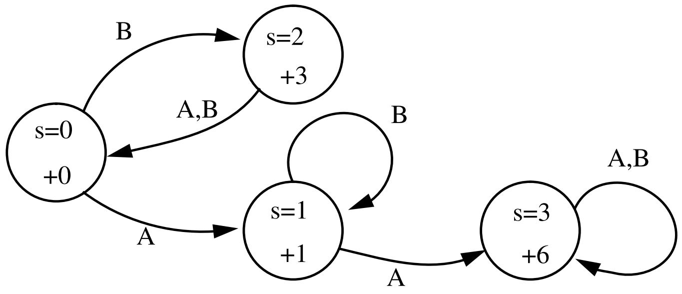

Figure 3: Example showing need for non-stationary policies

$t = 2$		$t = 1$	
state	*action*	*state*	*action*
0	A	0	B
1	A	1	A
2	A or B	2	A or B
3	A or B	3	A or B

Table 1: Policies for horizons $t = 2$ and $t = 1$

To see why a non-stationary policy is necessary for the finite horizon case consider Figure 3. Assume that our horizon is $k = 2$ time steps where $k = 2$ means that there are exactly two time steps for which the agent needs to make decisions. The optimal policies for $t = 1$ and $t = 2$ are shown in table 1. Notice that each is a policy (complete mapping from states to actions), but that they are different. Since the model is small it should be easy to convince yourself that any other policy would not do as well for each of the two time steps. Therefore the true optimal policy for a $k = 2$ horizon is to apply the $t = 2$ policy first and the $t = 1$ policy next. Note that optimality here is being measured as total accumulated reward for the two time steps.

The other class of problems we consider are termed *infinite horizon* problems. Although we wouldn't expect any agent to have to make decisions forever, we use this type of model when we do not know in advance how long the agent's lifetime will be, which in many circumstances is more realistic than the finite horizon models. At first, attempting to consider an infinite number of time steps might seem impossible. Although it does add some complexity, it can be and is often done, especially in the MDP model. The trick that needs to be incorporated is the addition of a discount factor to the rewards, so that rewards that are received further in the future are discounted more than rewards received closer to the present time. To show why the addition of the discount factor can makes things work out nicely, imagine that we have some divine knowledge of exactly which state we will be in at each step and which action will be executed at each step. We will represent this knowledge as the functions $S(k)$ and $A(k)$ respectively. With this knowledge we can write the value, in terms of rewards, as

$$V_t(s_i) = \sum_{k=0}^{\infty} \beta^k q_{S(k)}^{A(k)}.$$

This formula is associating a value for starting in a given state, s_i, with t time steps remaining assuming we will know exactly what states we are in, $S(k)$, and which actions we perform, $A(k)$, at each time step. Here k represents the time when there are $t - k$ steps remaining.

This formulation, usually referred to as the value function, is a power series of the immediate rewards. Since we ultimately want to discuss and compare the value of different policies we would like this series to converge. A well known property of power series of this form is that they will converge for variables with an absolute value of less than one. In this case the variable of interest is the discount factor β. Since it is difficult to interpret a negative discount factor in our problems, we will only consider discount factors in the range: $0 \leq \beta < 1$. A main motivation for using a discount factor is that it allows us to talk about and compare these infinite sums. The niceness of the mathematical formulation and convergence are hardly appropriate justifications for using a discount factor; however, more motivated interpretations and justifications of the discount factor will be presented later (as will the problems with using a discount factor.) There are also techniques for solving undiscounted infinite horizon problems [7, 14], but this paper does not attempt to treat these problems.

You will recall that, for the finite horizon problem, a policy had the form $\delta = \{\delta_0, \delta_1, \ldots, \delta_t\}$ since we (potentially) needed a different policy for each time step. We have thus far formulated our infinite horizon problem the same way, but trying to represent an infinite horizon policy this way leads to some real problems. First, how could we ever calculate an infinite number of policies and, even if we figure out how to do that, how could we put it into a form that we could use that wouldn't take up an infinite amount of space? This is one place where the infiniteness can actually help us out. Imagine for a moment that you knew δ_i, that is, the optimal policy to use at time $t = i$ when there is an unbounded number of decisions to be made in the future. Well then, at time $t = i + 1$ there is also an unbounded amount of time left as there is for any particular time. Therefore, the policy that is best must be δ_i since we defined δ_i to be the optimal policy to use when there is an infinite amount of time remaining. Since our infinite horizon solution looks like this: $\delta = \{\delta_i, \delta_i, \delta_i, \ldots\}$, we can simply refer to the policy as δ because at any time step we will always be using the same policy. This is what we previously termed a *stationary* policy, because at every time step the policy is the same. Solving the infinite horizon problem now looks a lot more promising, since we only need to search for and represent a single policy. Also, we have developed a way to ensure that all potential policies have a finite value which we can compare to one another.

2.4 Policies in MDPs

The goal of the techniques discussed in this paper is to derive a computational procedure for finding an optimal deterministic *policy* for a given POMDP model. Working toward that end, we first discuss policies in the MDP model and a traditional method (policy iteration) for finding optimal policies in these models. We will later generalize both of these to the POMDP case.

Since our aim is to find the optimal policy, we need a metric that gives us a measure of each policy's usefulness. This metric allows us to compare policies with one another. As a start in this direction, we define $V_t^{\delta_t}(s_i)$ to be the expected reward that the agent can accumulate in its lifetime if it executes policy $\delta = \{\delta_0, \delta_1, \ldots, \delta_t\}$ for t steps. When it is currently in state s_i, there are t steps for which the agent needs to make decisions given that it executes actions according to the policy δ_m at time m.

$$V_t^{\delta_t}(s_i) = \sum_j p_{ij}^{\delta_t(s_i)}[w_{ij}^{\delta_t(s_i)} + \beta V_{t-1}^{\delta_{t-1}}(s_j)]$$

Since our MDP model has uncertainty in the outcome of its actions, we use the probabilities of moving to all possible next states in a weighted sum of the rewards we would get if we actually moved into that state. The reward for moving to a state is the immediate reward received for the current action, current state and next state, w_{ij}^a, plus the value of state s_j with one less step remaining, $V_{t-1}^{\delta_{t-1}}(s_j)$. We discount the value of the next step by β, which will become necessary when we discuss the infinite horizon problems. For the undiscounted finite horizon problems we can just set $\beta = 1$. Factoring and substitution of formula 1 allows us to simplify this expression to

$$V_t^{\delta_t}(s_i) = q_i^{\delta_t(s_i)} + \beta \sum_j p_{ij}^{\delta_t(s_i)} V_{t-1}^{\delta_{t-1}}(s_j).$$

Note that for the non-stationary policies of finite horizon problems, this formula requires the value for the last $t-1$ steps before we can compute the value for the t^{th} step. Thus, the value function is computed with dynamic programming. The values for $V_0^{\delta_0}(\cdot)$ depend only on the q_i^a values. For the infinite horizon problem, the policy, $\delta($
$cdot)$, that gets executed at any time is the same. This results in a value function:

$$V^{\delta}(s_i) = q_i^{\delta(s_i)} + \beta \sum_j p_{ij}^{\delta(s_i)} V^{\delta}(s_j).$$

This is a system of $|\mathcal{S}|$ equations with $|\mathcal{S}|$ unknowns. As long as $0 \leq \beta < 1$, this system of equations will have a solution. Thus, whether we are solving the finite or infinite horizon problem, if we are given a policy (stationary or non-stationary), the value function formulas allow us to easily compute a metric for the policy. These value functions will allow us to compare different policies and eventually to prove that a given policy is optimal. These formulas show how we judge a policy's usefulness by the actions and states that the agent is led through.

In an MDP, any mapping from states to actions is a policy, but what we are concerned with is finding the optimal policy. Optimality is considered with respect to the value of the state-action pairs which are derived from the reward structure $\mathcal{W}$. We would like a policy that performs better than any other possible policy in terms of the value of all states the agent passes through. We will denote the optimal policy by δ^* and its associated value function by $V_t^*(\cdot)$. More formally, an optimal policy is one where, for all states, s_i, and all other policies, δ, $V^{\delta^*}(s_i) \geq V^{\delta}(s_i)$. It is non-trivial, [7, 3], that such a δ^* exists.

The distinction between the policy and the value function of a policy is a fairly important one. Some of the algorithms discussed in this paper construct both the value function and the policy, whereas others construct only the policy. Typically, it is easier to construct the policy than it is to construct the policy and its value function. However, given a policy the value function can be derived by the formulas above. Conversely, a policy can be constructed from the value function.

2.5 The POMDP Model

A partially observable Markov decision process, POMDP, is defined by a hex-tuple, $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \Theta, \mathcal{R}, \mathcal{W} >$. $\mathcal{S}$ is a finite set of states with state i denoted by s_i. $\mathcal{A}$ is a finite set of actions with action i denoted by a_i or sometimes just a when we refer to a specific action. $\mathcal{P}$ is the transition probabilities for each action in each state and defines the Markov process that the agent actually operates within, though it typically does not have access to this *core process*. p^a_{ij} denotes the probability that the agent moves to state s_j given that it was in state s_i and it just performed action a. Θ is a finite set of observations where θ_i denotes observation i. Although there is an underlying Markov process, the agent does not directly observe it. $\mathcal{R}$ is the observation model, in which $r^a_{j\theta}$ denotes the probability that we observe θ when we are in state s_j at time t and when our last action (at $t-1$) was a. $\mathcal{W}$ specifies the immediate rewards, $w^a_{ij\theta}$ denotes the immediate reward received at time $t+1$ for performing the action a in state s_i at time t, moving to state s_j at time $t+1$, and making observation θ. These immediate rewards are essential in our quest for optimal solutions, since it is these values that inform us how useful different strategies are.

There is another notation that will prove convenient in our later formulations. We let q^a_i be the immediate reward of performing action a in state i. It can easily be derived from the immediate rewards $w^a_{ij\theta}$ weighted by the transition and observation probabilities for the different actions and observations:

$$q^a_i = \sum_{j,\theta} p^a_{ij} r^a_{j\theta} w^a_{ij\theta}. \tag{2}$$

This will simplify many of the formulas that appear later in the paper. We have also used the q^a_i notation in describing the MDP model. The use is the same in that we would like it to stand for the immediate reward received for performing action a in state s_i, but the calculation differs since, in the POMDP model, we must factor in the uncertainty (probabilities) of the observations. Further use of the value q_i^a will not be confusing if you remember which model is being discussed.

There are actually two ways a POMDP problem can be formulated. In the first, which is the one presented above and used throughout this paper, observations are made after an action is taken. Alternatively, we could define a POMDP in which we first make an observation and then perform the action. These two formulations are essentially equivalent, since a problem in either one of these forms can be converted to a problem in the other. However, to avoid confusion we will always discuss the former since this is the model most often used in the existing literature.

2.6 Belief States

When we moved from a deterministic action model to a stochastic action model, we could still use the same basic machinery to solve the two types of problems. This was possible because we assumed the agent always knew where it was (or could be) in the finite state space. Thus, to find a policy the agent merely needed to decide what the appropriate action was for each state in this finite set. However, in the POMDP model, adding partial observability creates a real problem. If the agent can never know for sure what state it is in, how can it possibly know what action to take? One observation we can exploit is that the agent will be more likely to be in some states than others based upon the actions it has taken and the observations it has made. For instance, in Figure 4, even if we do not know whether we started in state $s = 0$ or $s = 1$, if we take action A and make observation $\theta = 1$, we are more likely to believe that we are in state $s = 3$ than $s = 2$, since the probability that we observe 1 in state $s = 2$ is so low.

It turns out that we can keep track of how likely we are to be in each of the states. We call this probability distribution over the state space the *belief state*. We denote the belief state $\pi = \{\pi_0, \pi_1, \ldots, \pi_{|\mathcal{S}|}\}$ where $|\mathcal{S}|$ is the number of states in the model and π_i represents the probability that we are currently in state s_i. An important point (and drawback) about maintaining a belief state is that we must know the model of the world if we wish to compute it.

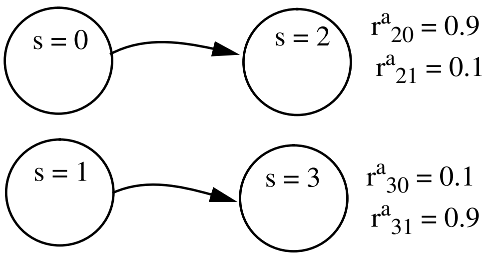

Figure 4: Belief state example

After each action and observation, we can update our belief state with a simple application of Bayes' rule:

$$\pi'_j = \frac{\sum_i \pi_i p^a_{ij} r^a_{j\theta}}{\sum_{i,j} \pi_i p^a_{ij} r^a_{j\theta}}. \tag{3}$$

The new belief state will be a vector of probabilities computed according to the above formula. We define a belief transformation function:

$$\pi' = T(\pi|a, \theta).$$

The first formula is more enlightening since it more explicitly states that we just consider all possible ways the agent could have ended up in a state weighted by the probabilities of those ways actually occurring for that action-observation combination. The second formula merely simplifies the notation for subsequent formulas. An important result, as shown in [16], is that this belief state captures all the necessary information for any sequence of actions and observations. Therefore, by constantly updating this belief state, we are implicitly saving the relevant part of our past history of actions and observations. Since we cannot know our location with certainty, this belief state seems like the next best thing.

A diagram of the basic dynamics of a POMDP is shown in Figure 5. The action is generated as a function of the belief state, and the observation is generated by the environment. Recall that we are using the POMDP model in which observations follow the actions.

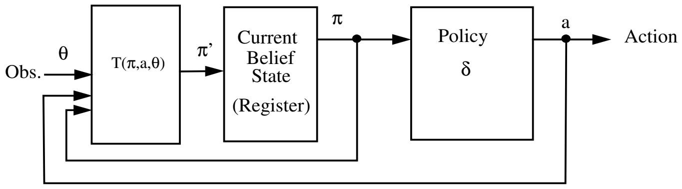

Figure 5: Control dynamics for a POMDP

2.7 Policies for POMDPs

In the regular MDP model, a policy is a mapping from states to actions and since the state space is finite, the policy and/or the value function are both easy to calculate and to represent (e.g., in a table). In the POMDP model, since we never know the true underlying state, our policies must now map belief states into actions. . The number of belief states is infinite, and therefore, storing the policy or value function in tables is no longer feasible. This means we must find some other representation for both the policies and the value functions of policies.

For now, we only consider the finite horizon case of a POMDP problem since this is the focus of much of this paper. A later section will discuss infinite horizon solutions.

Imagine that we need to solve the finite horizon for a POMDP problem with the horizon $k = 1$. In this case, the agent gets to choose to execute a single action and no more. Here, the observation it makes after that action doesn't really matter, because once it chooses an action the game is over. In order for this problem to be well defined, we need to define a terminating reward for the process. This terminating reward represents the value of the agent ending up in each of the environment states when $t = 0$ (i.e., there is no time left to take actions). We will denote the terminating reward for state s_i as q_i^0 Recall that there are also immediate rewards for each state-action pair. For $k = 1$, the agent will accrue one of these immediate rewards (depending on the state it was in and the action it chose to execute) plus the terminating reward for the state it ended up in. However, neither the starting nor ending state will be directly observable to the agent. All the agent will have access to is the probability distribution over the states (i.e., belief state) it starts out in. Our task is to derive a policy that obtains, based upon the belief state, the maximum expected reward for a single action.

Without knowledge of the actual underlying states of the agent, it might seem peculiar that we could ever find the best action to take. However, since the belief state represents the likelihood of being in each state, we can discuss the expected accumulated rewards by taking an expected value. For instance, assume that the agent's last belief state is π', then the expected terminating reward, or the value of ending up (at $t = 0$) in belief state π' is

$$V_0(\pi') = \sum_i \pi'_i q_i^0. \tag{4}$$

Now that we know the value of ending up in a particular belief state, we move on to discuss finding the best action for each belief state, π, given that we will only be executing one action. We know that the ending belief state will be $\pi' = T(\pi|a, \theta)$. The agent will have control over the action a, but not over the observation θ. Because it cannot control the resulting observation, the agent must use the model of observation probabilities to weigh all the possibilities:

$$V_1^*(\pi) = \max_{a \in \mathcal{A}} \Big[\sum_i \pi_i [q_i^a + \sum_{j,\theta} p_{ij}^a r_{j\theta}^a V_0(T(\pi|a,\theta))] \Big].$$

By substituting for $V_0(\cdot)$ and using formula 3 for $T(\pi|a, \theta)$, after some cancellation we get

$$V_1^*(\pi) = \max_{a \in \mathcal{A}} \Big[\sum_i \pi_i q_i^a + \sum_{i,j,\theta} \pi_i p_{ij}^a r_{j\theta}^a q_i^0 \Big].$$

With this formula we can determine the best action to take for any given belief state by performing the maximization shown. Thus, the optimal policy for $t = 1$ can be defined as

$$\delta_1^*(\pi) = \arg\max_{a \in \mathcal{A}} \Big[\sum_i \pi_i q_i^a + \sum_{i,j,\theta} \pi_i p_{ij}^a r_{j\theta}^a q_i^0 \Big].$$

Although the belief space is continuous, we are able to represent a policy over its entirety with this small finite formula, though there is some extra work required to perform the maximization. Notice here that the policy was derived from a value function. We show, shortly, that if you can represent the value function in this nice way then you also have a way to represent the policy since it is just the argument used in the maximization procedure. Indeed, this is exactly what we show for the general case next.

2.7.1 Value functions for POMDPs

To move to the general k-horizon case of the value function, we use the same ideas demonstrated for the 1-horizon case. First, let us assume we know the optimal value function when there are $t-1$ time steps remaining, $V_{t-1}^*(\cdot)$. The basic form is the same as the MDP, but here we have a few extra complications to consider. First, we must take the weighted average of all the states, with the weighting of a state's value coming directly from the belief state. Second, we need to factor in all the possible observations we could possibly make and take a weighted sum of the values using the observation probabilities. Lastly, when we consider the value for the next step we must transform the belief vector based upon the current belief state, the action and the observation. Folded together and taking the maximum for all possible actions we get

$$V_t^*(\pi) = \max_a \Big[\sum_i \pi_i \sum_j p_{ij}^a \sum_\theta r_{j\theta}^a \Big(w_{ij\theta}^a + V_{t-1}^*[T(\pi|a,\theta)] \Big) \Big].$$

Just as we did with the MDP model we can use factoring and substitution of formula 2 for the q_i^a values to simplify this to

$$V_t^*(\pi) = \max_a \sum_i \pi_i q_i^a + \sum_{i,j,\theta} \pi_i p_{ij}^a r_{j\theta}^a V_{t-1}^*[T(\pi|a,\theta)]. \tag{5}$$

This formula can appear rather complex at first. With its recursive definition and abundant summations it would seem that trying to solve for such a value function would get messy. However, if $V_t^*(\pi)$ is piecewise linear and convex (we show shortly that it is) it can be written much simpler as

$$V_t^*(\pi) = \max_k \sum_i \pi_i \alpha_i^k(t)$$

for some set of vectors $\alpha(t) = \{\alpha^0(t), \alpha^1(t), \ldots\}$. This is the crucial point for understanding the remainder of the paper and the algorithms. Unfortunately, this is also the easiest point to get confused. The simultaneous introduction of terms *piecewise linear and convex* and the $\alpha_i^k(t)$ notation can easily make your head spin at first. Going through each piece slowly, a piecewise linear function is one that is composed solely of line segments or hyperplanes. There may be many linear segments that together combine to make up this function, but at any point there is one line segment (hyperplane) that covers that point. A linear function of variables x_i can always be written with coefficients α_i as

$$\sum_i \alpha_i x_i = \alpha_0 x_0 + \alpha_1 x_1 + \ldots + \alpha_N x_N.$$

A piecewise linear function can consist of one or more linear segments of this form and so we superscript the coefficients of each to indicate which linear segment they come from so that the k^{th} linear segment will have the equation:

$$\sum_{i=0}^{N} \alpha_i^k x_i.$$

We will refer to the vectors $\alpha^k = [\alpha_0^k, \alpha_1^k, \ldots, \alpha_N^k]$ as α-vectors in the remainder of the paper and each vector will represent the coefficients of one of the linear pieces of a piecewise linear function. These piecewise linear functions will be the value functions for each step in the finite horizon POMDP problem. As a result it will be convenient to index the α-vectors by the number of time steps remaining where $\alpha^k(t)$ represents one of the linear pieces of the value function $V_t^*(\cdot)$.

We now develop the value functions for the finite horizon POMDP problem and show that they are indeed piecewise linear. With this result we can see how to represent a value function over a continuum of points (i.e., the belief space) with a finite number of items (i.e., α-vectors). This result is also shown in [17, 16]. This simpler representation is the key element to the POMDP solution techniques. Remember that the agent will only know the belief state, which is a probability distribution over the states of the model. With the MDP model there was a finite number of states and because the agent knew which state it was in, it could store the value function as a table. In the POMDP framework, the belief states are continuous. There are an uncountably infinite number of belief states which makes representing the value function by tables impossible. With this $\alpha(t)$ vector representation we have found a way to represent the value function for this continuous belief state space.

As an example Figure 6 shows a value function over an $|\mathcal{S}| = 2$ belief simplex. It consists of 4 vectors, which in theis case define 4 lines in a plane. Since our value function takes a maximum dot product of these for all belief points, the value function is the upper-most line segment at each point. This maximization always results in a convex function of this type for $|\mathcal{S}| = 2$.

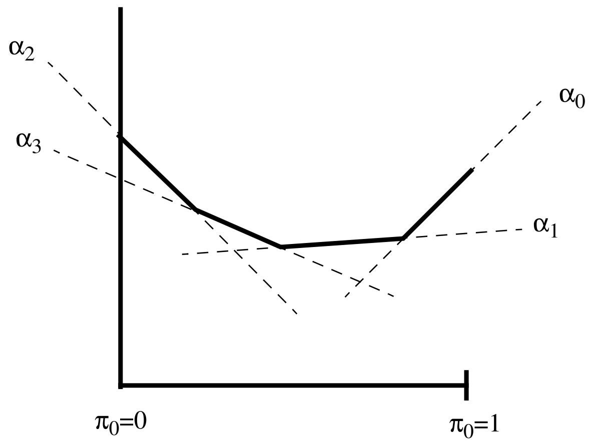

Figure 6: Sample piecewise linear and convex value function for $|\mathcal{S}| = 2$

This convexity generalizes for higher dimension state spaces as will be shown shortly.

Our value function above is defined recursively, but our vector representation of the value function is not, it merely shows the representation as if we knew the various α-vectors for that value function. To be useful, we would like a formulation that will define the vectors for time step t based upon the vectors from time step $t-1$. This is achieved by simple substitutions and the introduction of an indexing function to keep track of how we use the $t-1^{st}$ vectors, $\alpha(t-1)$. This will also demonstrate the inductive step in the proof that the value function for a finite horizon POMDP problem is piecewise linear and convex.

Our inductive hypothesis is that $V^*_{t-1}(\cdot)$ is piecewise linear and convex and thus can be written with some set of α-vectors, $\{\alpha^0(t-1), \alpha^1(t-1), \ldots, \alpha^m(t-1)\}$ as

$$V^*_{t-1}(\pi) = \max_k \Big[\sum_i \pi_i \alpha^k_i(t-1) \Big].$$

However, when we are interested in the value for time $t-1$, we will actually be dealing with a transformed belief vector so that we are really interested in

$$V^*_{t-1}(T(\pi|a,\theta)) = \max_k \Big[\sum_i \pi'_i \alpha^k_i(t-1) \Big].$$

Substituting for the transformed belief vector we get

$$V^*_{t-1}(T(\pi|a,\theta)) = \max_k \Big[\frac{\sum_{i,j} \pi_i p^a_{ij} r^a_{j\theta} \alpha^k_j(t-1)}{\sum_{i,j} \pi_i p^a_{ij} r^a_{j\theta}}\Big].$$

Every evaluation of this formula, for a specific π, a, and θ, can result in a different $\alpha^k(t-1)$ being selected (from the maximization process. Since we will soon need to put this definition of $V^*_{t-1}(\cdot)$ into the formula for $V^*_t(\cdot)$ we need a way to keep track of which of the $\alpha^k(t-1)$ was the maximum. To this end we define the function $\iota(\pi, a, \theta)$ to be the index of the α-vector that maximizes $V^*_{t-1}(T(\pi|a,\theta))$.

$$\iota(\pi, a, \theta) = \arg\max_k \Big[\sum_{i,j} \pi_i p^a_{ij} r^a_{j\theta} \alpha^k_j(t-1)\Big] \tag{6}$$

Notice that the π argument is actually being transformed by the function $T(\pi|a,\theta)$ and that the denominator is not relevant for the maximization procedure since it will be the same for all k. This is how the $V_{t-1}(\cdot)$ formula looks with our new index notation:

$$V^*_{t-1}(T(\pi|a,\theta)) = \frac{\sum_{i,j} \pi_i p^a_{ij} r^a_{j\theta} \alpha^{\iota(\pi,a,\theta)}_j(t-1)}{\sum_{i,j} \pi_i p^a_{ij} r^a_{j\theta}}.$$

The next step is to substitute this into our recursive definition of the value function given in formula 5:

$$V^*_t(\pi) = \max_a \sum_i \pi_i q^a_i + \sum_\theta \frac{[\sum_{i,j} \pi_i p^a_{ij} r^a_{j\theta}][\sum_{i,j} \pi_i p^a_{ij} r^a_{j\theta} \alpha^{\iota(\pi,a,\theta)}_j(t-1)]}{\sum_{ij} \pi_i p^a_{ij} r^a_{j\theta}}$$

After some summation manipulation, cancellations and factoring we are left with

$$V^*_t(\pi) = \max_a \sum_i \pi_i \Big[q^a_i + \sum_{j,\theta} p^a_{ij} r^a_{j\theta} \alpha^{\iota(\pi,a,\theta)}_j(t-1)\Big]. \tag{7}$$

The terms are arranged this way to show that we still have the value function in terms of linear segments. In this case the vector portion is defined by the inner bracketed quantity:

$$\alpha_i(t) = q_i^a + \sum_{j,\theta} p_{ij}^a r_{j\theta}^a \alpha_j^{\iota(\pi,a,\theta)}(t-1).$$

This completes the inductive step of the proof that the finite horizon POMDP problem value function is piecewise linear and convex. To complete the proof, we use the base case referred to earlier. The terminating rewards we defined in formula 4 show that the value function for $t = 0$ is also piecewise linear and convex. This proof was first shown in [17] as was the observation that the value function could be represented by a finite set of vectors.

If we know all the $\alpha^k(t-1)$ vectors for the $V_{t-1}^*(\cdot)$ value function, then by simple calculations using formulas 6 and 7 we can compute the above term to arrive at a linear segment, $\alpha(t)$, of the $V_t^*(\cdot)$ value function.

Another very important thing to notice about this new $V_t^*(\cdot)$ formulation is that for a given belief state π, in order to find its value we must do a maximization over all possible actions. According to the form above each action will result in a new vector. Each of these α-vectors for time step t has an associated action that gave rise to it. When we perform these maximizations in forming the $V_t^*(\cdot)$ value function we should also keep track of the specific action, a, that gave rise to it for each $\alpha(t)$ vector. With this combination of vector-action pairs we will have a compact way to represent the optimal policy for the finite horizon POMDP problem, namely just perform the maximization over all $\alpha(t)$ and take the associated action of that vector.

2.7.2 Geometric Interpretation of Value Function

The formulas and language of the previous section can serve to obscure the nice properties of the value function. In this section we attempt to improve the intuition behind the previously mention concepts of piecewise linear and convex and to provide a geometric interpretation of the value function. This representation is most valuable when trying to describe and understand the various algorithms discussed in this paper and in any other POMDP paper.

Note that with all of our previous definitions of value functions we have assumed that higher values are better than lower values. This stems from the fact that we chose the $w_{ij\theta}^a$ to represent rewards. However, the $w_{ij\theta}^a$ could be defined to represent costs. This formulation is found in some of the existing POMDP literature and results in the maximization procedures being replaced by minimization procedures. Other than this change the rest of

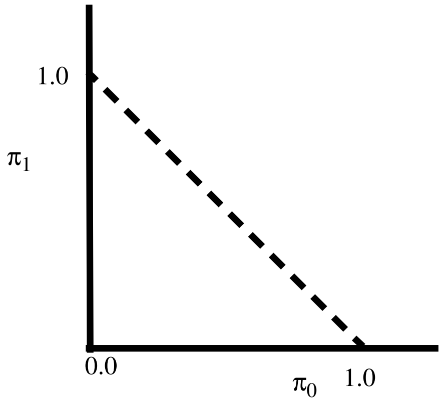

Figure 7: Simplex of belief space for $|\mathcal{S}| = 2$

the formulation remains unchanged except that typically the value functions $V^*(\cdot)$ are referred to as cost functions, $C^*(\cdot)$. We will continue to adopt the reward/value interpretation.

For all the following examples and diagrams we start with the space of beliefs. Since beliefs are probability distributions over the underlying model states we see the belief simplex for the 2 and 3 dimensional cases in Figures 7 and 8 respectively.

The value function is a function of the number of components in the belief state and is thus a function in $|\mathcal{S}|+1$ space where $|\mathcal{S}|$ is the number of states in the model. Because we would like to demonstrate these geometric ideas for the cases $|\mathcal{S}| = 2$ and $|\mathcal{S}| = 3$, we need a slightly different representation of the belief simplexes. This will prove to be both easier to understand as well as easier to produce on the printed page. The key to this other representation lies in the observation that one component of the belief state can always be omitted since the components (being a probability distribution) must sum to one. With this we can represent the 2 dimensional belief space as a single line or axis as in Figure 9. Here the distance from the left axis is the first component π_0 and the distance from the right can be used to compute the second component π_1. Our value function can now be represented by drawing it above this line. In Figure 9 we see a value function over the belief space which consists of two linear segments. Since we assume that we are concerned with getting the maximum reward the actual value function is always the larger of the two linear segments.

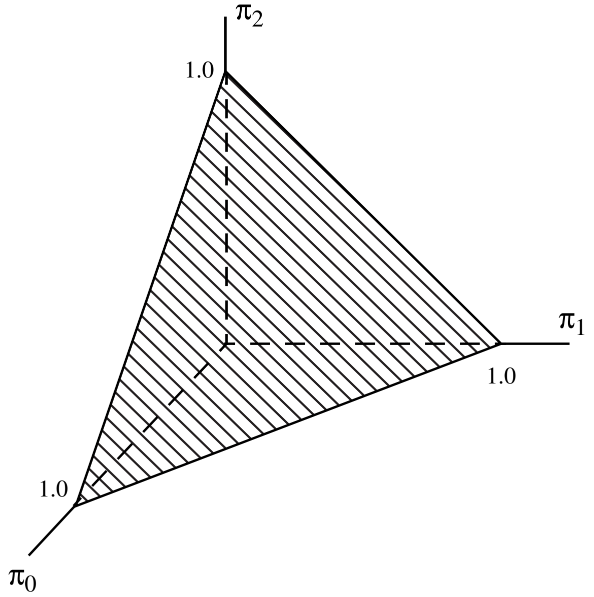

Figure 8: Simplex of belief space for $|\mathcal{S}| = 3$

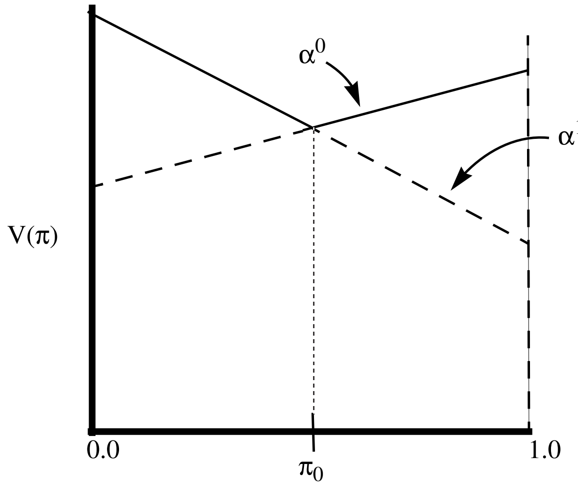

Figure 9: Sample value function for $|\mathcal{S}| = 2$

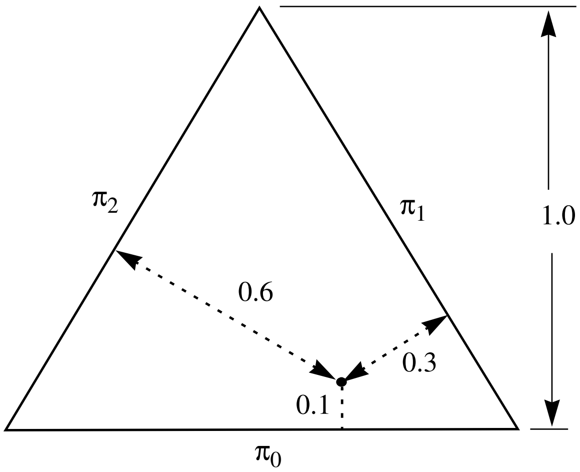

Figure 10: Example belief for $|\mathcal{S}| = 3$, $\pi = [0.1, 0.3, 0.6]$

Notice also that the value function for this case imposes a partition of the π_0 axis and thus is a partition of the $|\mathcal{S}| = 2$ belief space. This partition divides the belief space according to which of the linear segments (α-vectors) is maximum. The partition is shown with small vertical lines on the axis.

We turn the $|\mathcal{S}| = 3$ case into a two dimensional triangle as shown in Figure 10. In reality, the belief space is a two-dimensional triangle lying in three-dimensional space as shown in Figure 8, but in Figure 10 we view it simply as a two dimensional object where the value of a point in the belief space can be obtained by the perpendicular distance to sides of the triangle each distance representing a different component of the belief vector.

The value function for and $|\mathcal{S}| = 3$ problem can be thought of as a three dimensional surface lying above this triangle as shown in Figure 11. The surface is comprised of hyperplanes (i.e., it is piecewise linear and convex) and we can also view it as imposing a partition on the belief space. The borders of the partition are the projections of the intersection lines of the hyperplanes.

As shown previously, the value function is piecewise linear and convex. Geometrically this means that the value function is composes solely of straight lines (for $|\mathcal{S}| = 2$), planes (for $|\mathcal{S}| = 3$) or hyperplanes (for $|\mathcal{S}| > 3$). Each α-vector represents the coefficients for one of these lines or planes. The highest plane (or line) at a point is the α-vector that represents the value

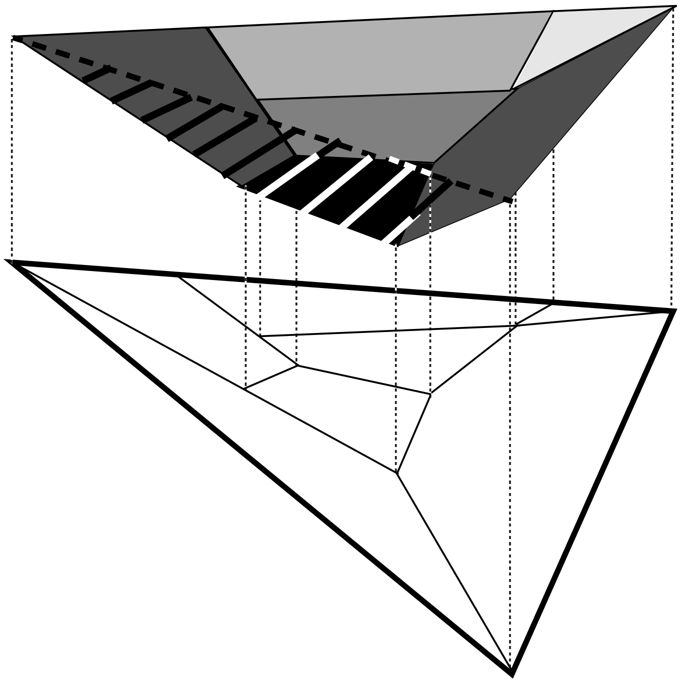

Figure 11: Sample value function for $|\mathcal{S}| = 3$

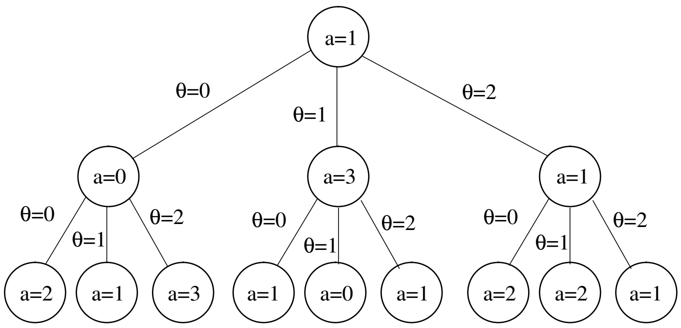

Figure 12: Example policy tree

function for this point. A key point with these pictures is that each α-vector defines a region over the simplex. These regions represent a set of belief states. Since there only a finite number of α-vectors, there are only a finite number of regions defined over the simplex.

2.8 Alternate Value Function Interpretation

There is another way in which we could go about determining the best policy for a k-horizon POMDP problem. This method comes from the observation that at each step there are only a finite number of things we can choose to do and a finite number of things we can possibly observe. Since we have a finite horizon, we could actually enumerate each possible policy we could follow as a decision tree. As an example consider the tree shown in Figure 12. This is one of the potential policy trees for the case where we have 4 possible actions, 3 possible observations and a finite horizon of $k = 3$. The nodes represent an action decision and the branch taken from a node depends on the observation made.

Knowing the initial belief state allows us to calculate the expected value for each of these trees. To do this we merely take an expectation for the immediate rewards (weighed by the belief state) for the action in the current belief state and then add the expectation for the subsequent actions. This requires us to compute the transformed belief states and weight each possibility by the probability of each observation. There are a finite number of trees and so we could just enumerate all of them. This would actually specify a policy since for a given belief state, we could evaluate all the possible trees to decide which is the best action to take. However, there are 4^{13} possible policy trees for this example since each of the nodes represents a decision point which could have any one of 4 possible values. The number of nodes in the tree will be determined by the formula

$$\sum_{i=0}^{k} |\Theta|^i = \frac{|\Theta|^{k+1} - 1}{|\Theta| - 1}.$$

and the thus the number of possible trees is

$$|\mathcal{A}|^{\frac{|\Theta|^{k+1}-1}{|\Theta|-1}}.$$

This is too many to expect to be able to generate. However, many of these policy trees will be useless in the sense that there will be no belief state for which they will be the best one. If we could somehow only generate the useful trees we would have reduced the complexity greatly. An additional savings could be made if we could collapse similar parts of the useful policy trees into the same sub-tree.

The the previous sections we have seen a way to construct a piece of the value function $V_t^*(\pi)$ given that we know the value function $V_{t-1}^*(\cdot)$. However, this only gives us the $\alpha(t)$ vector in the $V_t^*(\cdot)$ for a single specific belief state π. In order to create the entire $t = n$ value function we would need to apply formulas 6 and 7 at every single belief point in Π. There are too many of these points (an uncountably infinite number) for this naive algorithm (of applying these formulas to all points) to work. However, there will only be a finite number of possible $\alpha(t)$ vectors for any given horizon k. We will see that these $\alpha(t)$ vectors have a direct correspondence with the useful policy trees.

3 POMDP Solutions

We will first explore the structure of the solutions to finite horizon POMDP problems and their relationship with the solutions to infinite horizon problems. In the next section, we will present a number of algorithms that detail how to compute these solutions. A general method to solve the finite horizon problem is to iterate over time steps. This method first computes the optimal policy for the case when there is only one time step remaining. Given this solution, we now find the optimal policy when there are two remaining time steps. This will utilize the solution just computed for the one time step case. This process is repeated until we have arrived at the solution to the desired horizon size. This is a typical dynamic programming solution technique where the optimal solution for time step t is constructed from the optimal solution for time step $t-1$. For the finite horizon case we will actually be constructing the value function and the policy. For this, we will utilize the recursive formula for the value function which was described previously and is repeated here:

$$V_t^*(\pi) = \max_a \sum_i \pi_i \Big[q_i^a + \sum_{j,\theta} p_{ij}^a r_{j\theta}^a \alpha_j^{\iota(\pi,a,\theta)}(t-1) \Big].$$

Recall that the value function at each step can be represented by a set of vectors.

With this form we see that if we know how to compute the value function and optimal policy for one time step (given the value function for the previous time step as a set of vectors) then we also know how to solve the finite horizon problem for any horizon, since we can continually use the current solution as the input to the next iteration. Therefore, when we discuss these finite horizon methods we will only need to discuss the task of computing the value function, $V_t^*(\cdot)$, for one step given that we have, as input, the previous value function, $V_{t-1}^*(\cdot)$ as a set of vectors, $\mathcal{V}_{t-1}^*$.

One of the unpleasant parts of the finite horizon solution is that the resulting policy is typically fairly large. When we solve a k-horizon problem and perform the above mentioned iterative procedure, the final policy and value function we get out (the one for k remaining time steps) becomes only a small portion the actual solution. The policy and value functions calculated at each step are also part of the correct k-horizon non-stationary policy. The policy is non-stationary because we have a separate policy for each time step.

The reason the finite horizon solution is non-stationary becomes clearer if you consider how the agent would use a policy, knowing that it only has k time steps remaining. At first, there are k time steps remaining and so the agent can look at the policy it generated for this case (the last one computed in the iteration procedure). This will tell it the optimal action to take, but immediately after it executes the proper action there are $k-1$ time steps remaining. The policy it just used is not relevant for this case, however, in computing the k horizon policy we had to compute the $k-1$ horizon policy.

It is this policy that the agent needs to use now. This same argument applies for all the other time steps which requires retaining all of the intermediate work when solving the k horizon problem.

To head off a possibly confusing part of the notation we have been using, we need to emphasize exactly what the t^{th} time step represents. Here t is how many more decisions (actions) the agent has left to take. It is easy to interpret t as how many steps the agent has taken. This confusion can arise when one begins to think of the agent actually executing the computed policies. When we think of the agent acting with these policies, we would expect time to proceed forward (i.e., $t, t+1, \ldots$), but the policies it will execute will be proceeding in decreasing order of t, since after each step the agent has one less decision (action) to make.

We can use this same iteration procedure to solve infinite horizon POMDP problems. Although it isn't always possible to compute the exact optimal infinite horizon policy, we can get a policy that is as close as desired to the optimal. There are certain problems for which the optimal solution can be found exactly. These policies are called *finitely transient* and are discussed in a later section. With these type of policies, we can solve for larger and larger horizons until we find that two consecutive time steps have precisely the same solution. When this occurs, we know that further iterations will continue to give this exact same answer no matter how large the horizon is made. This follows because at each step we derive the optimal solution based solely on the previous optimal value function. If the previous optimal value functions are the same for any two consecutive time steps, then the computed optimal policy for these two times steps must be exactly the same.

The pleasant part about solving the infinite horizon case is that, unlike the finite horizon, we can discard all the intermediate results. It is only the final repeating optimal policy that needs to be retained since the optimal policy is a stationary one. Of course in order for the value function to converge, we must have a discount factor, γ, that is less than one. Otherwise, if we merely kept accumulating the rewards from all the previous time steps, the value function would grow without bound and never converge. The addition of a discount factor is a simple change to the value function:

$$V_t^*(\pi) = \max_a \sum_i \pi_i \Big[q_i^a + \gamma \sum_{j,\theta} p_{ij}^a r_{j\theta}^a \alpha_j^{\iota(\pi,a,\theta)}(t-1) \Big].$$

We will return to discuss the infinite horizon case in more detail later, but it is instructive to draw the connection between the finite and infinite horizon problems here.

An important point which has not yet been mentioned, though might be obvious, concerns the use of the value function. For one step of the iterated solution procedure, we assume that we have the previous value function $V^*_{t-1}(\cdot)$ with which to construct the new value function $V^*_t(\cdot)$. Our value function, $V^*_{t-1}(\cdot)$, will take the form of a set of vectors, $\mathcal{V}^*_{t-1} = \{\alpha^0(t-1), \alpha^1(t-1), \ldots, \alpha^{M-1}(t-1)\}$, where each vector is one of the linear facets that comprise the piecewise linear value function. As we saw previously, we can use the recursive value function equation to give us a new $\alpha(t)$ vector, but a key point is that in order to construct this vector, we must have a particular belief state, π. Using this, the value function formula for any valid belief state will give you an actual linear segment of the $V^*_t(\cdot)$ value function, however, it does not let you make any claims about the region (set of belief states) for which this is the optimal vector (other than it includes the belief state we used to generate it).

The bottom line is that given a point in the belief space, we can find a facet of the value function, but there is a problem because there are just too many (uncountably infinite) belief states for us to be able to crank all of them through the formula and construct the value function. One potential solution is to just randomly choose some finite subset of the belief points, push them through the formula and use the resulting set of vectors as the value function. One would hope that with enough points we would not miss any of the true vectors that comprise the optimal value function. This turns out to be a poor solution for many reasons. The first problem is to decide how many points to choose and this is directly related to the dimensionality of the belief space. The larger the belief space, the more points you would expect to need to choose. The worst problem with this approach (from the theoretical perspective) is that we can never guaranteed to find the optimal value function, since we never know whether or not there is some other point that would generate a different vector. We have run this approach on a number of problems and found empirically that it performs poorly, even for small problems.

The key to finding optimal policies (i.e., optimal value functions) is to develop a systematic (and terminating) algorithm to explore the entire continuous space of beliefs. A finite method is possible, since we know that there will only be a finite number of vectors to discover. The requires a more intelligent exploration and use of the belief space. The exact method used in exploring the belief space is where the algorithms discussed later in this paper differentiate themselves.

3.1 Structure of POMDP Solutions

The previous sections have discussed the mathematical foundations for solutions to POMDP problems. In this section we we look at the solutions from a practical view point. Although we do not discuss solution procedure until the next section, here we will begin to show how these solutions can be utilized. We first discuss the finite horizon problem and this will then lead us directly to the infinite horizon problem.

3.1.1 Finite Horizon

The raw output of the algorithms we discuss later is a set of vectors, $\mathcal{V}_t^* = \{\alpha^0(t), \alpha^1(t), \ldots, \alpha^{N-1}(t)\}$, one set for each decision time, t. This output refers to the finite horizon case where there is a non-stationary policy that requires all of the intermediate results. In addition, each of these vectors will have an action associated with it. To decide what action to take (i.e., use the policy) we take the set of vectors for the current time period and find the vector in this set which gives the largest dot product value with the current belief state. The action associated with this maximal vector is the action that should be executed. In this way, the policy can be stored as the set of vectors that comprise the value function along with an associated action for each vector.

In order to better motivate the discussion that follows we introduce a simple POMDP example, shown in Figure 13. In this example, we imagine that we are standing in front of two closed doors. Behind one of the doors is a tiger and behind the other is a large reward. If the door with the tiger is opened, then a large penalty is received (presumably as some amount of bodily injury). Most likely, we would prefer to open the door with the large reward instead. Aside from opening one of the two doors we have another action we can take, namely to *listen*. We choose to listen hoping that we will be able to hear which door the tiger is behind, but listening is not free, there is a cost associated with it. Unfortunately, listening is also not entirely accurate and there is a possibility that we will get the wrong information

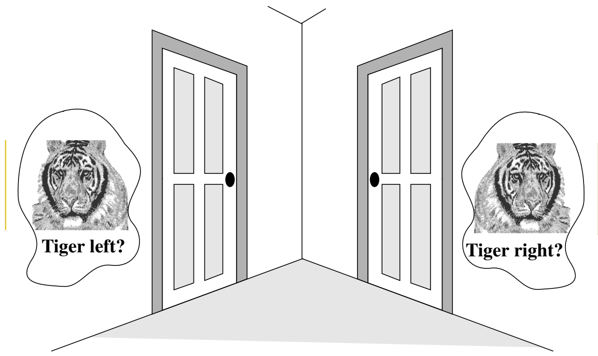

Figure 13: Example POMDP environment

when we listen. e.g., we hear a tiger behind the left hand door when the tiger is really behind the right hand door.

The complete set of parameters for this problem is given in Tables 2 through 4. State s_0 is used to represent the state of the world when the tiger is on the left and s_1 represents the state when the tiger is on the right. Action $a = 0$ is the action of listening and $a = 1$ and $a = 2$ are, respectively, the actions of opening the left door and opening the right door. The reward for opening the correct door is $+10$ whereas there is a penalty of -100 for choosing the door with the tiger behind it. The cost of listening is -1. There are only two observations possible; to hear the tiger on the left or to hear the tiger on the right. We have constructed this problem so that immediately after opening a door, and receiving a reward or penalty, the problem resets with the tiger being randomly relocated behind one of the two doors. Aside from resetting the tiger, the parameters have been constructed so that the belief state will be reset to the uniform distribution, $\pi = (0.5, 0.5)$, when either door is chosen, regardless of the belief state at the time the door is opened. This belief state is used to represent the situation where there is no a priori knowledge about the tiger's location.

If we are given the policy (i.e., the vectors and their associated actions) for each time step, then we can determine the proper action to take, but only if we have the current belief state. Let us begin with the policy for the time step $t = 1$, which is the policy when the agent will only get to make a single decision. There are three different actions the agent might choose to execute, open the right or left door or just listen. If our belief state was such that

p^0_{ij}	$j=0$	$j=1$
$i=0$	1.0	0.0
$i=1$	0.0	1.0

p^1_{ij}	$j=0$	$j=1$
$i=0$	0.5	0.5
$i=1$	0.5	0.5

p^2_{ij}	$j=0$	$j=1$
$i=0$	0.5	0.5
$i=1$	0.5	0.5

Table 2: Example transition probabilities

$r^0_{j\theta}$	$\theta=0$	$\theta=1$
$j=0$	0.85	0.15
$j=1$	0.15	0.85

$r^1_{j\theta}$	$\theta=0$	$\theta=1$
$j=0$	0.5	0.5
$j=1$	0.5	0.5

$r^2_{j\theta}$	$\theta=0$	$\theta=1$
$j=0$	0.5	0.5
$j=1$	0.5	0.5

Table 3: Example observation probabilities

$w^0_{ij\theta}$	$\theta = \{0,1\}$
$i=0, j=\{0,1\}$	-1
$i=1, j=\{0,1\}$	-1

$w^1_{ij\theta}$	$\theta = \{0,1\}$
$i=0, j=\{0,1\}$	-100
$i=0, j=\{0,1\}$	+10

$w^2_{ij\theta}$	$\theta = \{0,1\}$
$i=0, j=\{0,1\}$	+10
$i=0, j=\{0,1\}$	-100

Table 4: Example rewards

we had a high probability of being in the state "tiger-left". then we would imagine that the best thing to do would be to open the right door. In the symmetric case of having a high probability of "tiger-right", we would expect that opening the left door would be the best action to take. But what if we are highly uncertain about where the tiger is at this time? The best thing to do would probably be listening. To see why this is, notice that guessing wrong will cost us a penalty of -100, whereas guessing correctly only rewards us with 10. When we have no strong beliefs either way we would expect to guess wrong as often as we guess right. Thus, the expected reward would seem to be $\frac{-100+10}{2} = -55$. Listening always costs us -1 and so, in the long run, we would expect listening to cost us less that opening one of the doors in the case where we are uncertain about the tiger's location.

The previous paragraph sketches an intuitive argument for what the policy for $t = 1$ should be and it should be no surprise that this is exactly the optimal policy for this case. Figure 14 shows this policy pictorially. In this figure we show each of the vectors as a node in a graph (which currently has no edges). The actual vectors are shown above each node and below each node is the belief interval[1] over which it is the best vector. These three

[1]The belief interval is in terms of the component π_0 only since π_1 can be found by subtracting π_0 from 1.

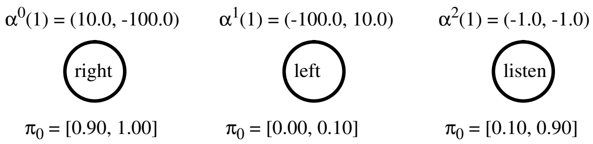

Figure 14: Tiger example policy for $t = 1$

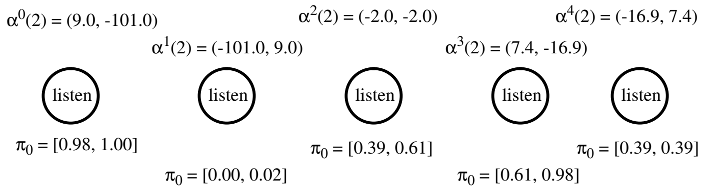

Figure 15: Tiger example policy for $t = 2$

vectors happen to cover the entire belief space (i.e., they form a partition of the belief space) and each has an associated action (shown inside the node circle) and so this specifies a policy for $t = 1$.

We now move on to the case where the agent has two time steps to make decisions for. The policy for $t = 2$ is shown in Figure 15 and has an interesting property; it always chooses to listen. There is a logical reason for this; if the agent were to open one of the doors at $t = 2$, then, due to the way the problem has been formulated, the tiger is randomly placed behind one of the doors and the agent's belief state will get reset to $\pi = (0.5, 0.5)$ (i.e., it has no information about where the tiger is). So after opening a door, the agent is left with no information about the tiger and one action left to take. We just saw that for the case where $t = 1$ and $\pi = (0.5, 0.5)$ the best thing to do was to listen. Therefore, no matter what happens, whether it opens a door or listens, one of the two decisions will result in a listen action. In a way, the problem parameters encode this and so the optimal policy will result in listening first, since this will give it information about the tiger. This way the agent will be better informed (since it has listened once) when it makes its next decision.

The other interesting feature found in Figure 15 concerns the number of vectors that exist. Although it will always choose to take the action listen, there are several vectors that have that particular associated action. These vectors are actually partitioning the belief space into pieces (areas or vol-

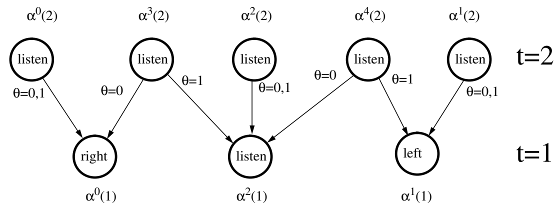

Figure 16: Belief state mapping from $t = 2$ to $t = 1$

umes) that have structural similarities. The similarity that the belief states in a particular partition element share is that when they are transformed, via $T(\pi|a, \theta)$ in Formula 3, the resulting belief states will all lie in the same partition defined by the policy for $t = 1$. In other words, every single belief state in a particular partition for $t = 2$ will, for the same action, a, and observation, θ, be transformed to a belief state that lies in exactly the same partition element that was imposed by the policy for $t = 1$. We have shown this relationship pictorially in Figure 16. Notice that the edges only need to be labeled with the observations since the action used in the transformation of the belief state is dictated by the node corresponding to that belief state, since each node represents a vector which has an associated action.

Interestingly, the optimal policy for $t = 3$ also consists solely of nodes with the listen action. The nature of the parameters of the problem are such that if it starts from the uniform belief state, $\pi = (0.5, 0.5)$, listening once does not give enough information (i.e., change the belief state enough) to make choosing to open a door more rewarding than listening. As mentioned before, choosing a door will always reset our belief state to this uniform distribution, therefore, paralleling the argument for $t = 2$, if we chose a door with $t = 3$ steps remaining, we would be guaranteed to listen for the next two steps. Thus, the optimal policy chooses the two listening steps up front, since that will give it the most information when it finally decides what to do at $t = 1$. You can see in Figure 16 that starting with belief state $\pi = (0.5, 0.5)$ we will choose to listen when $t = 2$ ($\alpha^2(2)$), and then, no matter what we observe, move to a belief state that lies in the listen node for $t = 1$ which corresponds to vector $\alpha^2(1)$.

This argument for doing the listening up front no longer applies after $t = 3$ and all the optimal policies for $t > 3$ will choose to open a door for some initial belief states. In Figure 17 we have shown the structure that

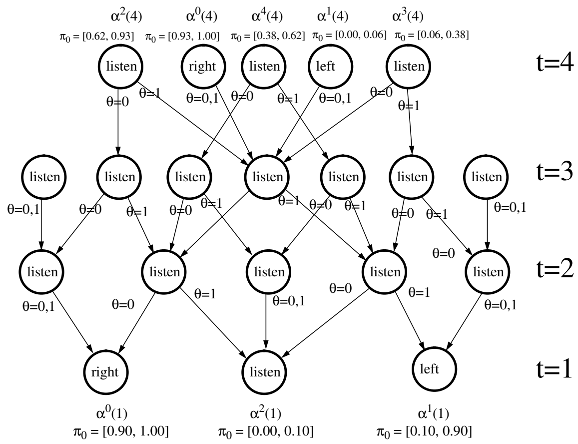

Figure 17: Policies and relationship for $t = 1$ through $t = 4$

emerges for the optimal policies from $t = 1$ to $t = 4$. Note that the belief state partitions imposed by these policies is only shown for the first and last policies.

Figure 17 shows many of the interesting features of a finite horizon POMDP solution. Notice that at the $t = 3$ level there are a couple of nodes, $\alpha^0(3)$ and $\alpha^1(3)$, that do not have any incoming arcs from the $t = 4$ level. This is interesting because it shows that no matter what belief state you start out in at time $t = 4$, there is no combination of action and observation that will leave you in a belief state that lies in the partition elements defined by those unused nodes at $t = 3$. (i.e., Certain belief states are not possible at this time step no matter what the agent's belief state was on the previous time step.)

Perhaps the most interesting feature of Figure 17 is the way in which the belief states behave. If we choose one of the belief partitions for $t = 4$, and execute this policy, the observations the agent makes will precisely define the nodes in the graph that would be traversed . Regardless of the actual starting belief state, as long as it lies in that particular partition element, the same action sequence will result for the same observation sequence, since the observations define which nodes to traverse and the nodes themselves define the actions to take.

The policy trees discussed in a previous section are very much related to this solution structure. Recall that a policy tree was a specific sequence of

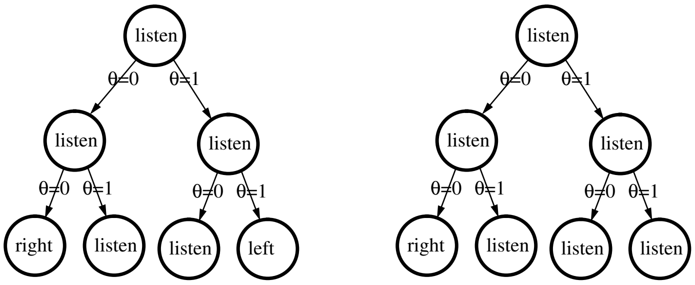

Figure 18: Two possible policy trees for tiger example

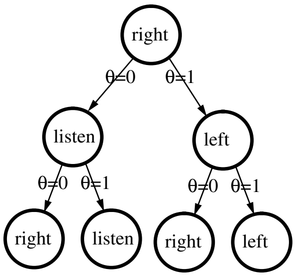

Figure 19: A policy tree that is not useful

actions to take based solely on the observations made. In that section we discussed the possibility of generating every possible tree and then evaluating each to see which was the best for a particular belief state. The graph shown in Figure 17 actually has some of these policy trees embedded in it. One major difference between this solution structure and the policy trees is that the solution structure does not have all possible policy trees in it, only the useful ones. The other major difference is that, within the solution structure, policy trees with similar sub-components are merged at these sub-components. For instance consider the two policy trees for $t = 4$ shown in Figure 18. These are two of the 2^{15} possible policy trees. Both of these policy trees are embedded in the graph of Figure 17 and hence are two of the useful policy trees. Notice how nodes of the tree are collapsed in the resulting figure. Figure 19 shows one of the policy trees that is not useful. There is no starting belief state where this tree will be the best thing to do and it does not appear embedded in the figure of the solution.

3.1.2 Infinite Horizon

With the finite horizon problems, the non-stationary policies that are generated can be quite cumbersome since there is a complete policy for each and every time step of interest. Without discounting future rewards, these policies will generate a different set of vectors (i.e., value function) for each and every time step.

When we add a discount factor to decrease the value of future rewards, the structure of the finite horizon POMDP problem changes slightly. As the time, t, gets larger the effect of the rewards received for early times (e.g., $t = 1$, $t = 2$, etc.) will begin to have negligible influence on the policy for later time steps. As $t \to \infty$, the vectors output by the discounted finite horizon algorithms begin to converge on a fixed set of values. As a result, for large t, the policy looks much the same as the policy for $t-1$. Figure 20 shows the solution to the *discounted* finite horizon of the tiger POMDP example for large values of t. Notice that the structure of the graph is exactly the same from one time to the next. The actual corresponding vectors for each of the nodes, which together comprise the value function, in the graph differ only past the fifteenth decimal place. This shows how the value function is converging. The structure of the graph remains unchanged, even though the values of the underlying vectors are slightly different from one time step to the next. This structure first appears at time step $t = 56$ and remains constant up to $t = 105$. When $t = 105$ the precision of the algorithm used to calculate the policy can no longer discern any difference between the vectors' values for succeeding intervals.

As we solve the finite horizon problem for larger and larger t, the value function (and its policy) for the largest time step gets closer and closer to the value function for the infinite horizon problem. The two converge when $t \to \infty$. Although in practice we cannot approach ∞, for any desired tolerance within the optimal, we can find a policy in a finite number of iterations. In this way we can use algorithms for the finite horizon problem to get arbitrarily closeto optimal for the infinite horizon problem. Since there is finite precision in most of the algorithm implementations, there will be a point at which the solution appears to converge. Although the actual value function only converges in the limit, for practical purposes, the solution the algorithms converge on for the discounted finite horizon are usually close enough to the actual infinite horizon solution. An advantage of infinite horizon over the

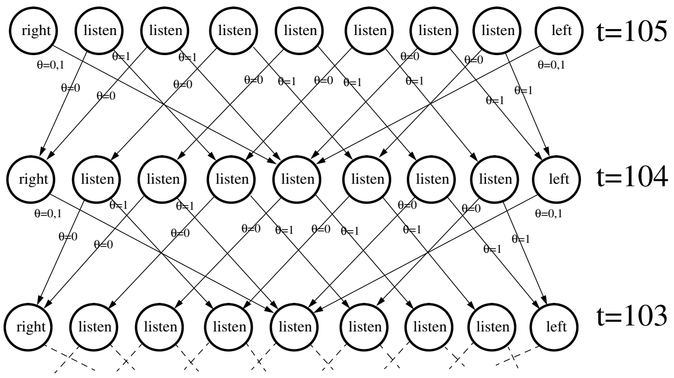

Figure 20: Structure of solution for large t

finite horizon solutions is that we only need to keep the last policy generated and we can throw away all the other time steps' policies, since the optimal solution for the infinite horizon problem is a stationary policy.

3.2 Policy graphs

One drawback of the POMDP approach is that the agent must maintain a belief state. While in theory this is not a significant obstacle, since we assume we know the model. in practice it can become a major problem when the agent might not have the resources (time or hardware) to perform the updating of belief states. Although updating the belief state is a trivial operation, when choosing which vector is maximum for a given belief point, we must dot product each vector with the belief state. This can become time consuming. Fortunately there is a way to encode the policy in a graph such that no explicit belief states need to be maintained and no dot products performed. We refer to such graph as *policy graphs* and they appear first in [17]. The policy graphs we discuss apply to the infinite horizon solutions.

Recall Figure 20 where we have converged upon an infinite horizon policy. If we continued to run the algorithm longer and longer, we would continue to get the exact same structure. Because we have the same structure at every level, we can re-draw the edges from one level to itself as if it was the succeeding level. Although this is erroneous in the finite horizon problem (because eventually we will get to lower values of t where the graph is not the same for consecutive time steps), for the infinite horizon, we can generate

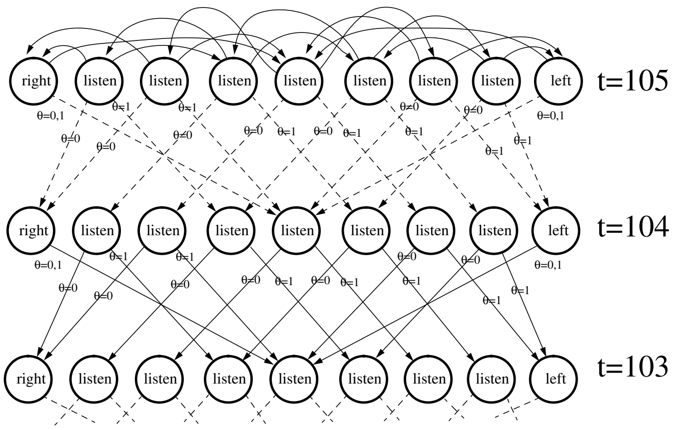

Figure 21: Rearranging edges for infinite horizon

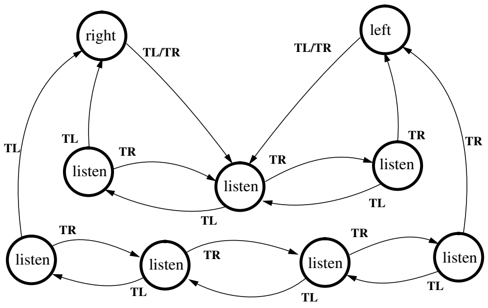

Figure 22: Policy graph for tiger example

as many of these identical levels as desired. This rearrangement of edges is shown in Figure 21.

In Figure 22 we have redrawn this graph into what we refer to as a *policy graph*. Here "TL" and "TR" represent the observations "tiger-left", $\theta = 0$, and "tiger-right", $\theta = 1$, respectively. The policy graph in Figure 22 has the interesting property that there are some nodes of the graph that will never be visited once either the open-left-door or open-right-door actions are taken. This results from the resetting of the belief state to $\pi = (0.5, 0.5)$ induced by the particular parameters of the formulation. If we imagine that the agent always starts in a state of complete uncertainty, then it will never be in a belief state that lies in a partition of these non-reachable nodes. This results in a simpler representation of the policy graph, shown in Figure 23.

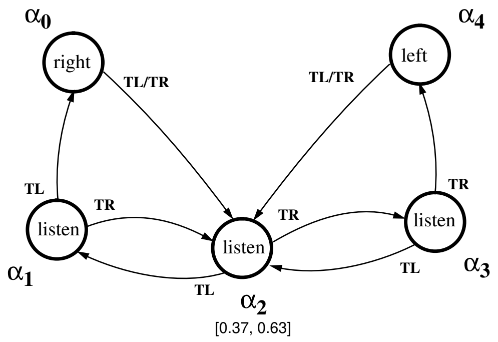

Figure 23: Trimmed policy graph for tiger example

Recall that the vectors that each node represents define a partition of the belief space and, furthermore, that these all belief states within a particular partition will map to the same node on the next level. In the case where we collapse the levels into one, we are guaranteeing that there is a function that maps partitions elements onto partition elements which is derived from the transformed belief states for a particular action and observation.[2] The policy graph representation allows us to forget about maintaining belief states, since as long as we are guaranteed that the belief states behave in this manner, the exact belief state is not important. The only thing that is important is which partition the belief state lies in and, by the properties of the policy graph, all the subsequent belief states will lie in partitions solely determined by the following the actions defined by the nodes of the graph and transitioning to another node based upon the observation.

3.3 Finitely Transient Policies

In our discussion about using finite horizon algorithms for finding solutions to discounted infinite horizon problems, we ignored a few important aspects about infinite horizon solutions. Solving the finite horizon for larger and larger horizon sizes will give solutions (value functions) that get closer and closer to the optimal infinite horizon problem. In some sense the solution will only converge on the optimal in the limit. Furthermore, the optimal value function might not be piecewise linear, though it will always be convex.

[2]This type of partition on the belief space is referred to as a *Markov partition* because the next partition element is solely determined by the current partition.

We developed the idea of policy graphs assuming that the solutions for each time step would eventually converge to some piecewise linear value function. However, this will only happen for a certain class of problems. Only when it does converge can we can construct the policy graph.

Finite Transience is formally defined in [17] and is the property of a policy not of the POMDP problem. Here, we opt for a more intuitive understanding. Recall that a particular policy will define a partition of the belief space and that each node of the policy graph represents one of these regions. Moving around in the policy graph (based upon the resulting observations), is actually the same as moving around in the belief space since the belief state will be transformed by $T(\pi|a,\theta)$ at each step (whether or not we actually choose to compute it). With the policy graph we are implicitly assuming that all the belief states within a particular partition element will be transformed to another element under $T(\pi|a,\theta)$ for a particular a and θ. This property held from the properties of the POMDP formulation and the structure of the resulting optimal policy. However, not all optimal policies for POMDPs will define partitions with this property, but the ones that do are termed finitely transient policies. For non-finitely transient policies we will not be able to construct policy graphs that are exactly optimal.

4 Finite Horizon Algorithms

4.1 POMDP History

POMDP research grew directly out of the MDP research, both of which began to flourish in the 1960's. [3] and [7] provided much of the basic framework and solution procedures for MDP models. Among the earliest work dealing with partial observability are [5] and [1]. Although none of these presents algorithmic solutions to the general POMDP model, each provided some of the groundwork for treating the general problem. In this sub-section we will give just an overview of some of the POMDP researchers and their work. Their algorithms are treated in more detail in the subsequent sub-sections.

The first researcher to give a detailed algorithm for finding optimal policies for the general POMDP model is E. J. Sondik [17]. In this work he gives an algorithm for finding the exact solution to finite horizon problems and solutions to infinite horizon problems that are arbitrarily close to optimal.

Sondik's thesis treats both the discounted and undiscounted infinite horizon, though this paper focuses only on discounted/undiscounted finite horizon and discounted infinite horizon by repeated finite horizon methods. Unfortunately, Sondik's finite horizon algorithm is difficult to follow as well as to implement and is slow for all but the smallest of problems. An attempt is made in a later section to present the details of Sondik's algorithm in a language that is more comprehensible than his thesis or subsequently published article [16].

Sondik's main contribution was to observe that, since there were a finite number of linear segments (regions over the belief simplex) in the value function, we could determine all of these segments by iteratively finding a particular segment and the belief space region for which it was optimal. Once this region was known you would know exactly where points in neighboring regions would lie, namely, on the borders of the current region. The algorithm was guaranteed to terminate since there are only a finite number of these regions in the finite horizon problem.

Monahan [12] presented a much simpler algorithm for computing optimal policies for the finite horizon. Although it is simpler to understand, at first it appears more inefficient than Sondik's method, because it is an exhaustive algorithm. However, for most problems it turns out to actually be more efficient than Sondik's method. The reasons for this are discussed in more detail in a later section. Monahan's approach is to enumerate all possible[3] linear segments that could exist for time $t = i$ and then go through them one by one to see which were relevant. Monahan's technique differs from most of the others presented since it does not explore the belief space in any way. His insight was that with a finite set of previous vectors, $\mathcal{V}^*_{t-1}$, and finite action and observation spaces, there were only a finite number of possible ways the value function could give different answers. Thus, Monahan decides to enumerate all possible $\alpha(t)$ vectors and check each for validity afterwards.

Eagle [6] presented the details of an optimization of this method, first suggested by Monahan, that reduces the work needed to solve the problem. Eagle's uses Monahan's observation that for many of the linear segments (α-vectors) enumerated by Monahan's method, many could immediately be

[3]There are actually an uncountably infinite number of possible vectors, but the structure of the problem will allow us to eliminate all but a finite number of them. It is these vectors that we consider the "possible" vectors.

dismissed if they were component-wise dominated by a previously or subsequently enumerated one. However, even with this optimization optimal policies to small problems can still require a substantial amount of computational resources. This technique still requires enumerating all the possible vectors, of which there are usually a large number.

Cheng introduced two algorithms [4] the first of which (Relaxed Region) is very similar to Sondik's except that he defines regions that are typically larger than Sondik's and thus, the algorithm tends to be more efficient. Cheng's other algorithm (Linear Support) uses successive approximations of the value function to converge on the optimal value function by comparing the current approximation with the optimal value function at a set of points. Cheng guarantees that if there is any difference between the two at all, then one of those points will show the maximum difference. An additional advantage of the Linear Support algorithm is that it can also be used to generate approximate solutions to any desired degree of approximation. Both of Cheng's algorithms require the use of an algorithm for finding all the corner points of the regions (i.e., interior points of convex polyhedra) which can result is poor running times.

Our Witness algorithm is similar to Cheng's Linear Support algorithm except that instead of having to discover all of the corner points of a region, we define a linear program to find points where the approximation differs from the optimal value function. Use of a linear programming formulation has resulted in empirically better running times.

At this time it appears that seeking exact solutions to anything but small POMDPs is not practical. There are many techniques for determining approximate methods and many of these are discussed in [9].

4.2 Overview of Algorithms

In the following sections we will give detailed descriptions of some of the better known algorithms for solving finite horizon POMDPs. We have ordered the algorithms in a slightly out-of-chronological manner. Our purpose here is to attempt to present the simplest two algorithms (Monahan's and Eagle's) first, both of which actually succeeded Sondik's method. After these we present, in chronological order, Sondik's One Pass algorithm, both of Cheng's algorithms (Relaxed Region and Linear Support) and finally our Witness algorithm. It is hoped that the presentations of these algorithms are clearer and more complete than some of the original descriptions of these algorithms. We conclude this section with a comparison of these existing techniques.

All of these algorithms have a few things in common. They all are performed iteratively (i.e., as a dynamic program). Additionally, all of the algorithms try to find the set of vectors that define both the value function and the optimal policy at each time step. The description of the algorithms is limited to the process of completing one iteration since each iteration requires the exact same technique; they all start with the finite set of vectors, $\mathcal{V}^*_{t-1} = \{\alpha^0(t-1), \alpha^1(t-1), \ldots, \alpha^{M-1}(t-1)\}$, for the previous time step (i.e., the piecewise linear segments of the optimal value function, $V^*_{t-1}(\cdot)$). They all produce another finite set of vectors, $\mathcal{V}^*_t = \{\alpha^0(t), \alpha^1(t), \ldots, \alpha^{N-1}(t)\}$, which represent the piecewise linear value optimal function, $V^*_t(\cdot)$, for the next successive time step. In addition, each of the vectors in $\mathcal{V}^*_t$ will have a single control action a associated with it. The $\mathcal{V}^*_t$ set comprises the value function, $V^*_t(\cdot)$, whereas the combination of the vectors and their associated actions are used in defining the policy, $\delta^*_t(\cdot)$.

The algorithms presented fit into two separate classes. One class starts with the set of previous vectors, $\mathcal{V}^*_{t-1}$, generates a superset, $\mathcal{V}^+_t$, of the actual vectors for the next step and then proceeds to reduce that set to the actual set, $\mathcal{V}^*_t$. The algorithms that fall in this class are those of Monahan and Eagle. In the other class of algorithms a slightly different approach is used. They begin with the previous set of vectors, $\mathcal{V}^*_{t-1}$, and construct subsets of the set, $\mathcal{V}^*_t$. Each subset can be viewed as representing an approximation to the optimal value function. These subsets grow larger and larger until the actual set, $\mathcal{V}^*_t$, is fully constructed. The advantage of the latter type is that they do not necessarily have to generate all possible vectors, of which there can be an extrememly large number. The Relaxed Region, Linear Support and Witness algorithms are in this second class.

All of the algorithms are presented as discounted finite horizon algorithms. The discounting factor β is not necessary for finite horizon problems, but is included for the case where we want to use the iterated finite horizon to solve for the discounted infinite horizon (in hopes that it converges). The discount factor is a trivial addition and does not make the algorithms any more or less complex. If fact, when $\beta = 1$, this is exactly the same as an undiscounted finite horizon problem.

It is easy to set up the iteration for all of these algorithms by feeding the resulting set of vectors, $\mathcal{V}^*_t$, back through the algorithm to get the set of $\mathcal{V}^*_{t+1}$ vectors. Note that the associated actions for the input vectors, $\mathcal{V}^*_{t-1}$, do not come into play for the algorithms when they are computing the $\mathcal{V}^*_t$ vectors. They are only required to define the policy, $\delta^*_{t-1}(\cdot)$, for the previous time step. Aside from the POMDP model and the initial set of vectors, no other input is needed for these algorithms (except the discount factor if one is being used).

4.3 Finding a Vector for a Single Belief Point

This is one step that is common to all of the algorithms and so we have chosen to deal with it in a separate section. Given a single point in the belief state space we can use

$$\alpha_{a,i}(t) = q_i^a + \gamma \sum_{j,\theta} p_{ij}^a r_{j\theta}^a \alpha_j^{\iota(\pi,a,\theta)}(t-1) \tag{8}$$

to immediately generate a vector for each possible action. This formula actually only shows how to construct a single component of the vector, so it must be repeated for each component i. For these vectors (referred to as $\alpha_a(t)$) we can use the formula

$$V_t^*(\pi) = \max_a \Big[\sum_i \pi_i \alpha_{a,i}(t) \Big] \tag{9}$$

to find out which of these will be the true vector at the belief point π. This maximal $\alpha_a(t)$ then is one of the true linear segments of the value function. Note that this is just the inner quantity from the recursive value equation from formula 7.

Being able to generate the $\alpha_a(t)$ vector immediately might be a bit of an over statement, it really involves a few steps, but each step is straight-forward and requires nothing more than simple calculations. However, we will step through this exact procedure since it is a crucial step in all of the algorithms. There is also an important complication involving ties in the values that is dealt with at the end of this section.

The basic scheme is that we will need to try all possible values of action a in the formula and then choose the one that gives the maximum value. An important point here is that it is not enough to merely save the action a and its maximal value (from formula 7), you must also save the inner bracketed quantity (which is actually a vector), since once you find the action that maximizes the value function it is the corresponding inner quantity that is the α-vector for the particular belief π.

In the value function above, the quantities q_i^a, p_{ij}^a, $r_{j\theta}^a$ and β are directly available from the POMDP model and the belief state π is assumed, for the moment, to be given. The only unspecified term appears to be $\alpha^{\iota(\pi,a,\theta)}(t-1)$. If you recall, the function $\iota(\pi, a, \theta)$ was merely a convenience that was used to succinctly represent the previous value function, $V_{t-1}^*(\cdot)$. It was defined as

$$\iota(\pi, a, \theta) = \arg\max_k \left[\sum_{i,j} \pi_i p_{ij}^a r_{j\theta}^a \alpha_j^k (t-1) \right].$$

We see in this formula that everything we need to compute $\iota(\pi, a, \theta)$ is readily available: p_{ij}^a and $r_{j\theta}^a$ from the model and $\alpha_j^k(t-1)$ as the input to the current iteration. So we see that given a particular belief π, we need only to crank it through the above two formulas and arrive at a piece (one of the linear segments) of the optimal value function, $V_t^*(\cdot)$.

Since we know there are only a finite number of these vectors (linear segments or α-vectors), if it were possible to know the exact finite set of points that would generate each of these vectors, we could construct the entire optimal value function as a set of α-vectors from a finite set of points. Unfortunately, this set of points is not readily available and some work must be expended to determine these points. All the algorithms discussed except one, try to be clever about finding this set of points. The exception, Monahan's algorithm, is conceptually much simpler because it does not seek to find this set of points.

4.4 Monahan's Algorithm

We start with the easiest algorithm of the bunch, though it was presented after the landmark work of Sondik ([17] [16]). Monahan actually credits this algorithm to Sondik and, though there are similarities, Monahan's description of Sondik's algorithm has enough differences to warrant a separate treatment. Monahan's algorithm is both easy to understand and easy to implement. However, do not expect to be able to solve anything but the smallest of problems with this algorithm. An important part of this algorithm's description is that it provides a lot of the groundwork that will be essential for understanding the other algorithms.

We begin with our optimal value function

$$V_t^*(\pi) = \max_a \sum_i \pi_i \Big[q_i^a + \gamma \sum_{j,\theta} p_{ij}^a r_{j\theta}^a \alpha_j^{\iota(\pi,a,\theta)}(t-1) \Big].$$

The key insight behind Monahan's method is noticing that there are only a finite number of possible vectors (i.e., inner bracketed quantities) that can be constructed from the above formula, despite the fact that there is a continuum of belief states. The actual values (components) of the $\alpha_a(t)$ vectors generated are not dependent upon the value of the belief state chosen. The belief state serves only to decide which of the $\alpha^k(t-1)$ vectors to include in the summation over the observations. Since there are only a finite number of previous vectors in $\mathcal{V}_{t-1}^*$ and a finite number of both actions and observations, we can simply list all the possible ways that the value formula could be constructed. Sure, there can be a good number of them[4], but the number is a finite one and we could enumerate and calculate all of them, given enough time. Notice that since we have abandoned considering the belief state, π, we cannot really attempt a maximization over the different actions, since the maximization is based on the value of the function which is in turn dependent upon the particular belief state.. Instead, we add this into the enumeration scheme, so that now we consider for each possible action all the ways the $\alpha^k(t-1)$ vectors could be combined in a summation over the observations, θ.

4.4.1 Monahan's Enumeration Phase

Here we will try to be more explicit about Monahan's enumeration scheme so that no confusion can arise. For a given action, as we do a summation over the observations, θ, we get a choice of one of the previous $\alpha(t-1)$ vectors for each θ. This amounts to filling in a table for each action (e.g., $\theta_0 = 4$, $\theta_1 = 2$, $\theta_2 = 9$, etc.) where the numbers assigned to each observation are one of the $\alpha(t-1)$ indices (of which there are $M = |\mathcal{V}_{t-1}^*|$ possible choices). This means that there are $M^{|\Theta|}$ ways we could fill out this table and that there are at most $M^{|\Theta|}$ ways to do the summation over observations. However, each summation can apply for each action and so there are at most $|\mathcal{A}|M^{|\Theta|}$ possible new $\alpha(t)$ vectors we could generate. The first step of Monahan's method is to create all of these possible vectors.

[4] $|\mathcal{A}|M^{|\Theta|}$, where $M = |\mathcal{V}_{t-1}^*|$.

4.4.2 Monahan's Reduction Phase

The biggest problem now is that our value function probably doesn't require all of these new $\alpha(t)$ vectors. We have enumerated them without concern for whether or not they really provide a maximal value for some belief state. Chances are that a significant number of them will not be maximal for any belief state. We could just keep them all, and each time we need to find what action to perform, maximize over all of these and select its appropriate action. However, this is a lot of excess baggage to carry around since we could waste a significant amount of time checking vectors that will never be maximum regardless of the belief state chosen. For time critical applications, it would not be very desirable to have the policy stored this way. An even worse problem arises in the finite horizon problem, since we iterate many times sending the results from one iteration to the next. If we never trim away the useless vectors, the number of vectors in the next iteration might be an order of magnitude larger than if we had kept only the relevant ones. Even when we do the trimming, the number of vectors at each iteration grows quickly, without trimming this problem is greatly magnified. A later section provides some analysis about how large these sets can grow, here we are content to say that there are a whole lot of them[5].

To trim away the useless vectors generated during the enumeration phase, Monahan's method utilizes linear programming (LP). Linear programming is also used in most of the other algorithms discussed. What linear programming does is more important than how it does it, but for the reader interested in how LPs work see [19]. There are many different ways to view what is happening in a linear program. We opt for the geometric view, which is more natural given that we have been trying to present the belief space and value functions geometrically. Geometrically, a linear program defines a region in some space. This region is defined through a set of constraints in the LP which are generally inequalities, but which can be equalities. There is an objective function, which is a linear function defining a line/plane/hyperplane through the space. The process of solving a linear program will attempt to either maximize or minimize the value of this objective function within the defined region.

The observation that will lets us use LPs to trim the set of vectors gen-

[5] $|\mathcal{A}|^{\frac{|\Theta|^{k+1}-1}{|\Theta|-1}}$

erated during the enumeration phase, is that for a vector to be a useful part of the optimal value function, there must be at least one belief state, π, for which it gives a larger value than all the other newly created $\alpha(t)$ vectors. Imagine that we want to check if $\alpha^j(t)$ was a true vector. To do this we set up a linear program with a constraint for each $\alpha^k(t)$ we have generated:

$$\sum_i \pi_i \alpha_i^k(t) \leq \sum_i \pi_i \alpha_i^j(t), \qquad \forall k.$$

This formula states that we are only interested in the region of the belief space where $\alpha^j(t)$ will provide a better value[6] than all other $\alpha^k(t)$. The variables in this LP are the components of the state vector, π_i, and a rewriting of this in a more useful way is:

$$\sum_i \pi_i(\alpha_i^k(t) - \alpha_i^j(t)) \leq 0, \qquad \forall k.$$

We have to add one additional constraint to these, namely

$$\sum_i \pi_i = 1.$$

This makes sure that the point lies on the belief space simplex (i.e., it is a probability distribution). Most LP packages assume that all of the variables are non-negative. If this is not the case then we must also add the constraints for the ranges of the variables

$$\pi_i \geq 0, \qquad \forall i.$$

We can use virtually any objective function we want with this LP, since here we are only concerned with whether or not there is any feasible solution to the LP. If there is any solution at all, then there must be some point on the belief state simplex that satisfies all of these constraints and thus $\alpha^j(t)$ is indeed part of the optimal value function. If the LP is infeasible, then not only have we found that it is not part of the optimal value function, we no longer need to keep it around for subsequent LPs.

The linear programming formulation and solving described above only checks a single vector. This entire process must be repeated for each of the

[6] As stated, the constraint merely states that it provides at least as good a value as all other vectors. This can lead to problems which are discussed in the next sub-section.

$\alpha(t)$ vectors enumerated. The only pleasing part of this procedure is that the size of these LPs will diminish as we are able to trim more and more useless vectors, since once we have determined a vector to be extraneous, we don't have to generate a constraint for it when checking the other vectors. The whole algorithm then becomes:

```
(1) Generated all possible vectors from formula 8.
(2) Add each vector to a list and mark the vector.
(3) Choose a marked vector from list.  If none exist,
  then we are done, the list contains all useful vectors.
(4) Otherwise, construct the LP for that vector (see above)
  and unmarkthe vector.
(5) If the LP is not feasible, then remove the vector
  from the list.
(6) Go to step (3).
```

4.4.3 Monahan's LP Complications

When we solve one of the LPs described in the previous section and find that it is feasible, we are guaranteed that the particular vector under consideration is at least as large as all the others for at least one point. This would seem to be exactly what we are looking for, but things are not quite this simple. It is possible for a vector to satisfy these conditions and not really have any significant use in the value function being constructed.

Figure 24 shows a simple two dimensional case of a vector that is not really useful. From point a to b, vector α_1 dominates (i.e., is greater than or equal to) all the other vectors. The same holds for vector α_2 for points between b and c. However, notice that vector α_3 dominates at the single point b. This vector isn't really useful, because both of the other vectors, α_1 and α_2, give just as high a value as α_3, but a linear program, set up as described in the previous section, will ignore this fact. The LP will only concern itself with whether or not there is at least one point where α_3 gives at least as large a value as the other two. In this case point b satisfies this condition, though there are not any other points that do. What we need is a way to formualte the LP so that we only get vectors that have at least one

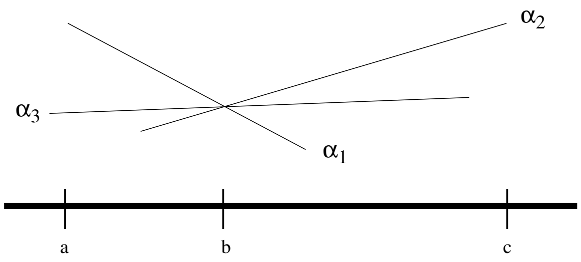

Figure 24: Example of a useless vector found by an LP

point where it is greater than all the other vectors.

What we would really like to do is to change the LP so that instead of having *less than or equal* constraints we have strict inequalities. Unfortunately this is not feasible within the linear programming framework. Instead we utilize a trick to get a similiar effect by introducing an extra variable, δ, into the LP. To each constraint's left hand side we add δ. If we can force δ to be greater than zero, then it must be the case that all of the left-hand sides, as they would have appeared in the old LP formulation, are strictly less than zero. To force δ to be greater than zero, we make the LPs objective function δ and attempt a maximization. We then solve the LP in the normal way and check the value of δ. If it is greater than zero, then the vector under consideration is one that is not useless. Otherwise, we have a vector of the type shown in Figure 24 (though probably a higher dimensional equivalent). As usual, if the LP is not feasible we have a truly useless vector that we can eliminate.

To summarize the addition of the δ variable, the new Monahan LP formulation for a vector, $\alpha^j(t)$, is:

$$\max : \delta$$

$$\sum_i \pi_i(\alpha_i^k(t) - \alpha_i^j(t)) + \delta \leq 0, \qquad \forall k.$$

$$\sum_i \pi_i = 1.$$

4.5 Eagle's Variant of Monahan's Algorithm

Monahan [12] mentions that dominated vectors could be removed to help reduce the number of linear programs that need to be solved. In [6] this optimization is made more explicit. The optimization occurs in the phase where we need to enumerate all of the possible new $\alpha(t)$ vectors. Eagle's shows that if, in our enumeration process, we ever come across a vector whose components are completely dominated by another vector's components (one already generated by the enumeration process), then we can immediately discard it since it is impossible for it to be a true vector in the value function. In other words, if we just generated $\alpha^j(t)$ and the following condition holds:

$$\exists k, \forall i, \qquad \alpha_i^k(t) \geq \alpha_i^j(t)$$

then we can immediately discard $\alpha^j(t)$. This is true because all the belief components π_i are non-negative. Finding a single existing vector that is component-wise larger than another implies that for any belief π at all, the former will result in a larger value, thereby rendering the latter extraneous.

Aside from checking only to see if a newly generated vector is dominated by one that has already been enumerated, we can also check to see if the new vector dominates any of those have been previously enumerated. The same domination argument applies for removing these vectors from the current list. In summary, we check the new vector to see if any in the current list dominate it. If so we discard the vector, otherwise we then check to see if any vectors in the list are dominated by the new vector. We remove any from the list that are dominated and finally add the new vector to the list.

Other than this check during the enumeration, Eagle's variation works exactly the same as Monahan's: enumerate all possible vectors and verify each one with a linear program. Eagle also mentions that we can skip the linear program verification step if we are willing to pay the computation price that will be incurred by keeping the extraneous vectors around. Since each extra vector in one time step can lead to many extra vectors in the next step, this seems to be an impractical solution for any problem. It is unlikely that the any LP package would be slow enough to prefer eliminating this step.

4.6 Sondik's One-Pass Algorithm

4.6.1 Background

Edward J. Sondik [17] presented the first solution techniques for finding optimal policies for general POMDP problems. This was the seminal work from which all the other algorithms described in this paper were derived. His algorithm for the finite horizon case is also described in [16]. This latter reference can be both a help and a hindrance. The formulas and notation in this article are much easier to follow than that of his thesis, and his article is devoid of the many theorems and proofs that serve to complicate the issues. However, a few typographical errors, a couple of vague sentences and one serious oversight [13, 9] make this exactly the wrong primary source to use for an attempted implementation.

4.6.2 The One-Pass Algorithm

This algorithm begins where all the other ones begin; with the recursive value function:

$$V_t^*(\pi) = \max_a \sum_i \pi_i \left[q_i^a + \gamma \sum_{j,\theta} p_{ij}^a r_{j\theta}^a \alpha_j^{\iota(\pi,a,\theta)}(t-1) \right]$$

Here, unlike the exhaustive enumeration of Monahan, Sondik actually develops a method for finding the proper set of belief states to plug into this formula to get all the necessary vectors. Remember, there are a finite number of vectors and we could generate them all if we only knew which belief states to plug into the formula. Furthermore, his algorithm guarantees that this set of belief states is finite and that they generate all the true vectors comprising the value function (i.e., none are missed). Unfortunately, the size of the set of belief states Sondik's algorithm needs to check is typically significantly larger than the number of vectors that need to be discovered.

This algorithm amounts to a search through the belief space by moving from one region to another until every region is found. We will soon discuss what exactly these regions represent, but here the important point is that his algorithm is guaranteed to only visit a finite number of regions and that the union of all these regions is the entire belief space (i.e., these regions are a finite partition of the belief space.)

The basic idea behind the algorithm is not too complicated to state, but trying to implement this algorithm exposes many subtle problems. Dealing with these problems is what adds the real complications to an implementation of this algorithm. Here is an over-simplified version of Sondik's algorithm.

(1) Initialize a search list of belief states to contain
  any single point on the belief simplex. Initialize
  a set of vectors, $\hat{\mathcal{V}}_t$, to be empty.
(2) Remove a point from the search list. If the list is
  empty, then we are finished and $\hat{\mathcal{V}}_t = \mathcal{V}_t^*$.
(3) Find the true vector (and its associated action)
  for this point and add it to $\hat{\mathcal{V}}_t$. (Use formulas 6 and 7)
(4) Define a region around this point where this vector
  is guaranteed to be the true vector.
(5) Select points that lie on the edges of this region
  and add them to the search list.
(6) Go to step (2).

Although the algorithm has been greatly simplified thus far, it is nonetheless an important abstraction to understand, since it is the very heart of this and the subsequent algorithms. Before we discuss the complications that can arise, we will first expand upon the steps as they are currently shown.

The first two steps are straightforward and require nothing more than maintaining a list-type data structure to store belief vectors (which are just tuples of size $|\mathcal{S}|$ and $\alpha(t)$ vectors (which are also tuples of size $|\mathcal{S}|$, but also having an associated action with each). A previous section describes, in detail, how to perform step (3), just remember that in addition to generating the vector you need to store the associated action that was used to create it, since the action is important in constructing the policy.

Steps number (4) and (5) are the most interesting. These are where Sondik provided real insight into POMDP problems. As mentioned earlier, when we generate a vector from a single belief point, we have no guarantee about what the true vectors are for all the other points in the belief state space. However, this isn't quite true. Although information about other belief points is not explicitly available, there are some ways in which we can extract this information. Sondik, by examining the steps used to create a vector, was able to define a region (volume) around this belief point where that vector is guaranteed to be the true linear portion of the value function. His method is to construct a series of constraints on the belief states that, when satisfied, assure that the belief states satisfying those constraints have this vector as the maximal one. This has the effect of defining a volume of belief states instead of the single point we started with. Describing the region, step (4), is necessary for generating the new belief states to add to the search list, as is done in step (5). We first describe the former step.

A little extra notation is required in order to develop the constraints that define the regions. We use π to represent the point selected from the search list. Recall that to find the true maximum $\alpha(t)$ vector we need to do a maximization over all actions in formula 9's maximization procedure. What this really gives us is $|\mathcal{A}|$ vectors for the point π where $|\mathcal{A}|$ is the cardinality of the set of actions, i.e. number of possible actions. We will refer to each of the vectors generated in this step as $\alpha_a(t)$ where a is the action that generated the vector. We use $\alpha_{a,i}(t)$ to represent the i^{th} component of those vectors. Of these $\alpha_a(t)$ vectors, one of them will give the maximal value for the particular belief π. We will refer to this vector as $\alpha^*(t)$. Additionally, the action for this best vector will be referred to as a^*. Note that $\alpha^*(t) = \alpha_{a^*}(t)$.

We now must think about how the belief state π could vary, while still resulting in $\pi \cdot \alpha^*(t)$ being larger than all $\pi \cdot \alpha_a(t)$. This exact condition is actually one of the constraints on the region we are looking for:

$$\sum_i \pi_i \alpha_i^*(t) \geq \sum_i \pi_i \alpha_{a,i}(t), \qquad \forall a.$$

Notice that this formula exactly expresses our desire to constrain the belief states to satisfy the condition stated before with one constraint for each $\alpha_a(t)$. However, this is just the tip of the iceberg as far as constraints are concerned. All of the $\alpha_a(t)$ were generated dependent upon the actual value of π. Although the above constraints hold true when all the $\alpha_a(t)$ remain unchanged, variations in π can cause changes in the $\alpha_a(t)$ vectors. Therefore, the above set of constraints is too liberal, since it might include belief states where the $\alpha_a(t)$ vectors have completely different values. We need to constraint the belief state further to ensure that the actual values of the $\alpha_a(t)$ stay the same. To do this we must examine the formulas that gave rise to the $\alpha_a(t)$ vectors.

There are two ways in which our $\alpha^*(t)$ vector could result in a lower value than one of the other $\alpha_a(t)$ for a particular π. The first is if the components of the $\alpha^*(t)$ vector change. If these components change, we can no longer be sure that it will dominate all of the other $\alpha_a(t)$ vectors, since it will depend upon exactly how the components changed and by how much. The other way $\alpha^*(t)$ can lose its domination over the other $\alpha_a(t)$ vectors is if one or more of the $\alpha_a(t)$ vectors change. In this case, although the quantity $\sum_i \pi_i \alpha_i^*(t)$ remains unchanged, the quantities we compared it to have changed. Again, whether or not $\alpha^*(t)$ still provides the best value depends upon how the $\alpha_a(t)$ vectors changed. There is also the case when both $\alpha^*(t)$ and the $\alpha_a(t)$ vectors change simultaneously, but handling the individual cases is sufficient for dealing with this situation.

What affects the $\alpha_a(t)$ and $\alpha^*(t)$ values? The $\iota(\pi, a, \theta)$ function does. Recall that this function was a convenience for specifying the index of the previous time step's vector to use in the value formula, $V_t^*(\cdot)$. The $\iota(\pi, a, \theta)$ function represented a maximization of the dot product of the transformed belief state, $T(\pi|a, \theta)$, with all the $(t-1)^{st}$ vectors. All we need to do is to constrain the belief states so that this maximization works out the same as it does for π, and we will have ensured that the $\iota(\pi, a, \theta)$ function does not change. If the $\iota(\cdot)$ function stays constant, then the $\alpha_a(t)$ vectors that are generated in the maximization remain the same, which is the condition we are interested in. We restate the formula for the maximization over previous vectors so that the origins of the constraints to follow are easier to see:

$$V_{t-1}^*(T(\pi, a, \theta)) = \max_k \sum_{i,j} \pi_i p_{ij}^a r_{j\theta}^a \alpha_j^k(t-1).$$

Let ι be the index k that makes this maximum for a particular point, π, action, a, and observation, θ. Then the constraint

$$\sum_{i,j} \pi_i' p_{ij}^a r_{j\theta}^a \alpha^\iota(t-1) \geq \sum_{i,j} \pi_i' p_{ij}^a r_{j\theta}^a \alpha^k(t-1), \qquad \forall k \neq \iota$$

restricts the belief points, π', to a region of the belief space where $\iota(\pi, a, \theta)$ function does not change. This equation actually specifies a series of constraints, one for each vector in $\mathcal{V}_{t-1}^*$. Also, the above constraint only restricts one combination of a and θ and does not ensure that the entire function, $\iota(\pi, a, \theta)$, remains unchanged. To ensure that every part of the function does not change, we need a set of these constraints for each combination of a and θ. All of these constraints combine to ensured that neither $\alpha^*(t)$ nor any of

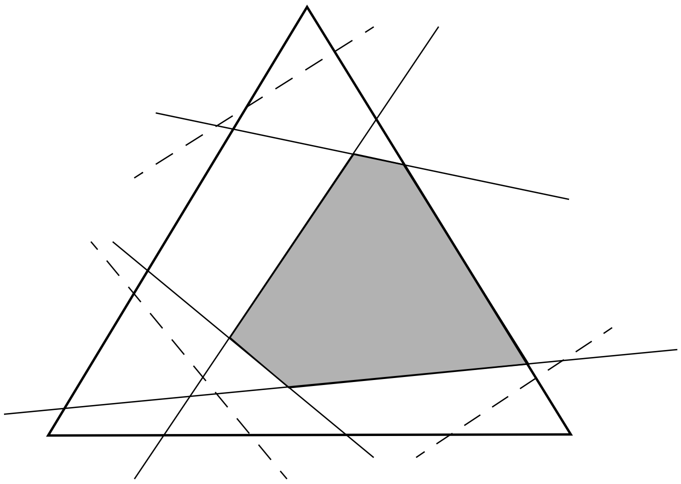

Figure 25: A belief space region defined by a set of constraints

the $\alpha_a(t)$ change. Due to the nature of the problem, the constraints defined only specify a region where we are ensured that things do not change. They do not provide any information about how things change outside this region. In fact, things might not change at all for points outside this region. This leads to defining regions that can be smaller than the actual regions formed by the various vectors in the optimal value function, $V_t^*(\cdot)$.

Aside from the constraints thus far outlined, there are also the constraints that restrict the belief states to lie on the belief state space simplex, namely

$$\sum_i \pi_i = 1$$

and

$$\pi_i \geq 0, \qquad \forall i.$$

Geometrically, Figure 25 shows the definition of a region through these constraints. This is a simple two-dimensional representation where each line represents one of the constraints' boundary lines. Since each is an inequality, a constraint actually consists of a region with all the points on one side of the line. The union of all these regions is the belief space region we have been discussing and is shown shaded in Figure 25.

No doubt, this looks like a formidable number of constraints and, it is for this reason that this algorithm has not found wide-spread use in solving

POMDP problems. The complexity of this algorithm is discussed further in a subsequent section, but the worst case number of regions for one time step is $|\mathcal{A}|M^{|\mathcal{A}||\Theta|}$ where $M = |\mathcal{V}_{t-1}^*|$. Each region requires a linear program of size $|\mathcal{A}||\Theta|M + |\mathcal{A}| + 1$ constraints and $|\mathcal{S}|$ variables.

After much work, step (4) is now complete; we have defined a region of the belief space where $\alpha^*(t)$ is guaranteed to be the optimal value function vector. This is especially pleasing because we have, in effect, eliminated every belief state within this region from consideration. We now know that for every belief state in this region, if we crank it through the value function we will get exactly $\alpha^*(t)$ and thus, there is no reason to consider these points further.

Although most of the conceptual work was done in step (4), the majority of the computational work lies in step (5). In this step we want to find belief states that are guaranteed not to be in the region we defined in step (4). If we can find a point not in the step (4) region, then we have found a point that must lie in some other region. With this point, we can do exactly as we did in step (4) and find its region, and so on until a complete partition of the belief space is found. When we have been to every region in the partition, we are sure to have generated all of the true vectors in the value function.

Given the region in step (4) how can we find points in other regions? The answer lies with linear programming. We have already built up the constraints of a linear program; notice that they were all linear constraints in the π_i variables. We rearrange the constraints above into more standard LP constraints and cluster them here for convenience:

$$\sum_i \pi_i \left(\alpha_{a,i}(t) - \alpha_i^*(t)\right) \leq 0, \qquad \forall a \neq a^*$$

$$\sum_{i,j} \pi_i p_{ij}^a r_{j\theta}^a \left(\alpha^k(t-1) - \alpha^{\iota(\pi,a,\theta)}(t-1)\right) \leq 0, \qquad \forall a, \theta, k \neq \iota(\pi, a, \theta)$$

$$\sum_i \pi_i = 1$$

$$\pi_i \geq 0, \qquad \forall i.$$

To find points in neighboring regions we are interested in points lying on the edge of the region defined by the constraints. These linear constraints will define a piecewise linear convex region (volume) of the belief space. The neighboring regions will lie directly adjacent to this region and will share

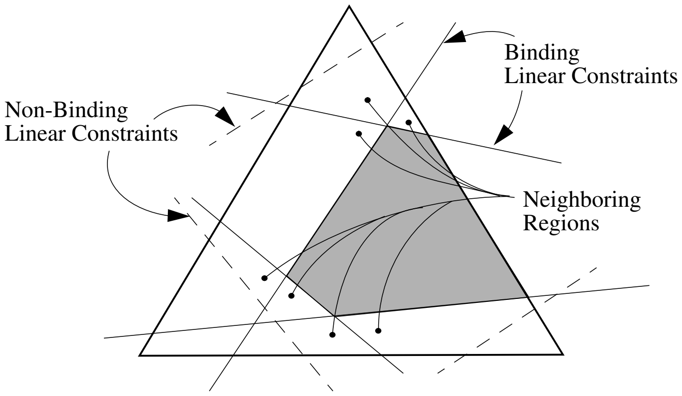

Figure 26: A region and its neighbors

either a common border (hyperplane) or a common point. Points on the borders of the currently defined region will also be points on the border of the neighboring regions. Figure 26 shows this graphically.

Since our current region will typically border many other regions, we need to find a point on each region border. Although we have a large number of constraining equations, there is a small subset of them that actually define the region. In Figure 26 the solid lines show those constraints that define (are binding on) the region and the broken lines are constraints that are superfluous (i.e., there are other constraints that restrict the size of the region more severely that it). The problem is that we have no direct way to ascertain which constraints are binding. The method proposed by Sondik to find the binding constraints is to solve a series of linear programs. There are actually two ways this can be done. We present both, first the one proposed by Sondik and then an alternative.

In Sondik's approach to finding binding constraints, a single LP is solved for each constraint in the LP (aside from the simplex constraints). The constraints for all of these LPs are identical (the ones shown previously), the only thing that changes is the objective function. For each constraint to be checked we use the constraint itself (without the right hand side, 0) as the objective function. Note that this constraint remains in the set of constraints for the LP. Solving this LP as a maximization problem will result in an answer that assigns values for all the π_i variables. However, just because this LP was feasible and had a solution point, it does not necessarily mean that we have found a new belief state to add to our search list. We only want to add this point if the constraint currently under consideration is binding in this LP. Fortunately, the solution to the LP gives us that information as well. In an LP, each constraint has an associated slack variable which indicates how much tolerance we have in changing this constraint without affecting the solution. If the constraint that was inserted into the objective function has no slack (i.e., its slack variable has value zero), then the constraint is binding and thus, we can insert the resulting value of π_is into our search table. In summary, for each constraint in the LP defined in step (4):

```
(1) Make constraint the objective function.
(2) Solve the LP trying to maximize the objective
  function.
(3) Check the slack variable of row that the constraint
  in the objective function appears.
(4) If it is zero then the solution (values for all πi)
  should be added to the search list, otherwise the
  constraint is non-binding.
```

A slightly simpler method for finding which constraints are binding, that still utilizes one LP for each constraint, is to slightly change the constraints as each LP is computed. In this method the actual objective function does not matter. What we do for each constraint (again, except for the belief simplex constraints) is to change its inequality into an equality. The result of solving this LP gives all the information desired. If the LP has any solution at all, then the constraint must be one of the binding constraints because a non-binding constraint cannot pass through the region defined by all the other constraints. Additionally, any solution it returns must lie directly on the border that this constraint makes for the region. With this method it doesn't even matter if we perform a maximization or minimization of the objective function and we do not need access to the slack variables (which aren't always readily available from an LP package.) The only trick here is to remember to change the equality back into an inequality when you move on to try the next constraint.

One aspect of the solutions to the LPs that needs to be pointed out concerns the nature of the solutions they return. With an LP formulation,

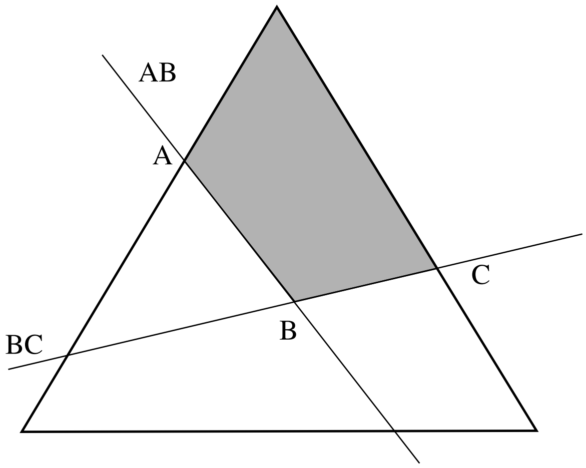

Figure 27: Points found by linear programs

the solution can consist of no points (i.e., an infeasible LP), one point or an infinite number of points[7] An LP solution with an infinite number of solutions happens when all these points result in the same finite value for the objective function. The nature of the LPs we solve, except in a rare instance, all have an infinite number of solutions (if it has any at all), however the answer the LP gives (assuming that it uses some form of the simplex method) will be a single point (one from the infinite set) and furthermore it will be one of the corner points or vertices of the region.

To illustrate this, in Figure 27, we see a region that has three adjacent regions that it shares a border with. In this case there will be two binding constraints we will call AB and BC. When we check constraint AB we find that there are really an infinite number of points that lie on the border (the entire line segment AB. However, the LP will return either A or B. Exactly which one is returned depends upon the exact nature of the LP and the way the LP package is implemented. For the constraint BC we could get either point B or point C. At first this might seem troublesome since it will be possible for these two constraints to result in the same point B, but this in itself is not a problem since all we need to guarantee is that we get a point that is on the border of a neighboring region. Point B satisfies this criterion

[7] An LP with an infinite number of solutions is different from an unbounded LP. In an unbounded LP the objective value function can be made arbitrarily large.

for all three of the neighboring regions. This raises some serious questions about handling these borderline cases. This is all discussed further in the next section.

This completes the description of Sondik's basic algorithm. It may or may not have raised some questions at each step. Hopefully, many of these questions (which have, thus far, been deliberately ignored) are answered in the next section. Trying to deal with the complications while describing the algorithm would have been difficult to do without adding much confusion. However, trying to implement the algorithm based solely on the previous description is not recommended. There are too many unanswered questions that will necessarily arise when an implementation is attempted.

4.6.3 The Complications

For all the lengthy description, the basic ideas behind Sondik's algorithm are really not that complex. However, there are some fairly nasty subtleties lurking beneath the surface. They all emanate from overlooking the case where we can get identical values as we perform the various maximizations in the value function while calculating the $\alpha(t)$ vectors. Recall that there are actually two places we need to perform a maximization: once over all the actions in the actual value function itself (over all the various $\alpha_a(t)$ vectors); and the other as we determine the function $\iota(\pi, a, \theta)$ (over all the previous $\alpha^k(t-1)$ vectors). It is possible (and very likely) that, when performing these maximizations, we get two or more instances that provide exactly the same maximal value. When there are ties or this sort, we cannot choose arbitrarily one from among the candidates, this <u>will not</u> work.

We will first try to present what is happening geometrically for the two cases of ties discussed above. Let us first suppose that we have generated all of the $\alpha_a(t)$ for a particular belief state π. Our next task is to perform the maximization to find which a and $\alpha_a(t)$ give the largest value for $\sum_i \pi_i \alpha_{a,i}(t)$. Let a' and a'' be two actions and furthermore, assume that both of their associated vectors give the same maximal value in formula 9. Then this indicates that we are on the border of two separate regions as shown in Figure 28. We might be tempted to arbitrarily choose one, say $\alpha_{a'}(t)$, with the rationale that the $\alpha_{a''}(t)$ region is a neighbor and thus will be found when we set up the region for $\alpha_{a'}(t)$.

But what happens if we do set up the region for $\alpha_{a'}(t)$ and our LP returns

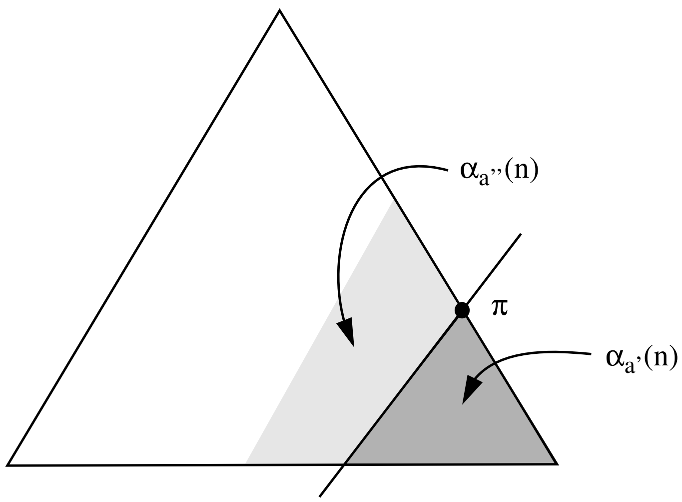

Figure 28: A point on the border of two regions

π as the point on the border between the two regions? Trouble. We would either see that we have used this point already and ignore it, or we might perform the same maximization calculation at this point and again choose the $\alpha_{a'}(t)$ region. In the former case the algorithm could completely miss the $\alpha_{a''}(t)$ region and, in the latter case, the algorithm could cycle forever on this point π. Although you might think that a little extra bookkeeping can keep you out of this trouble (e.g., just remember what choice you make at each point), consider the situation shown in Figure 29. This is an instance where a point is on the boundary of four regions. This requires more than a little bookkeeping, and as the dimensionality of the problem increases, this problem can get arbitrarily complicated. Constructing a truly general solution to handle these cases is possible, but it is cumbersome and prone to errors.

The examples above only dealt with the case of ties in the maximization over the actions in the value function of formula 9. We still must deal with the maximization over the previous vectors (formula 6) that give rise to the $\iota(\pi, a, \theta)$ function. The regions depicted above were somewhat misleading since they seemed to show the regions of the optimal value function, $V_t^*(\cdot)$ being constructed. In reality, the regions defined by Sondik's constraints are usually only a subset of the optimal value function vectors' regions. To discuss the next complication, we need to examine these regions closer.

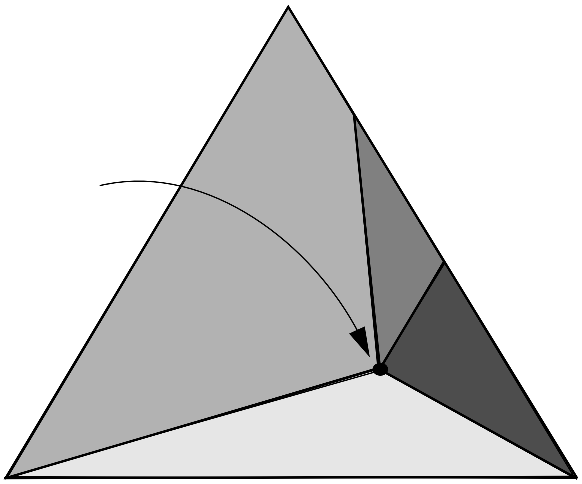

Figure 29: A point on the border of four regions

The constraints that define the region specify exactly the conditions necessary for the $\alpha_a(t)$ and the $\alpha^*(t)$ vectors to change. The unpleasant part of this is that these conditions are not necessarily sufficient for a change. What this implies is that we can tell when some of the vector components might change, but we cannot determine whether they actually will change or even if this change will have any noticeable affect on the maximal vector. It is possible for the components of some of the $\alpha_a(t)$ vectors change, while $\alpha^*(t)$ remains unchanged, but $\sum_i \pi_i \alpha^*(t) \geq \sum_i \pi_i \alpha_{a,i}(t)$ may still hold for π outside of the region defined by the constraints. The bottom line is that the constraints built up by Sondik's method are typically too conservative. The regions (or volumes) defined are usually proper subsets of the actual regions (volumes) for a given vector. Figure 30 shows the typical relationship between the true $\alpha^*(t)$ value function region and the region defined by the constraints.

Therefore, in reality, when we draw the regions that the algorithm generates we cannot really label it as the $\alpha^*(t)$ region since, in all likelihood, it is merely a subset of the true region of the value function. How then, could we assign a unique indentifying label to these regions? To answer this we should consider how many possible ways we can construct a set of constraints, since this represents the maximum number of regions we could possibly define. The constraints that arise from a region are fully determined by the optimal

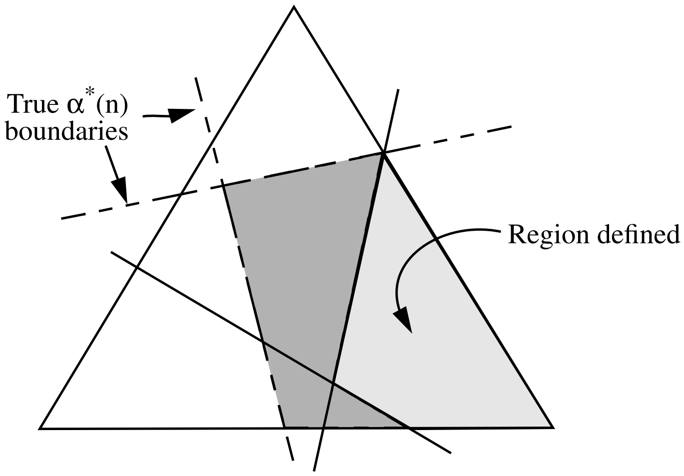

Figure 30: Sondik's region vs. actual region

action and the $\iota(\pi, a, \theta)$ function. Notice, the actual value of the belief state π never appears in the LP for the region (though it does help determine what the $\iota(\pi, a, \theta)$ function should be.) This observation shows that Sondik's algorithm can potentially explore a very large number[8] of regions, even when there are a small number of actual regions imposed by $\mathcal{V}_t^*$. The complexity of this is discussed in more depth in a subsequent section, but here we needed just to demonstrate the connection between the $\iota(\pi, a, \theta)$ function and the LP regions. Now we have established that each instance of an $\iota(\pi, a, \theta)$ function and action can, potentially, specify a different LP region in Sondik's algorithm. It might be the case that many instances of this function specify the same region, but the only way to determine that is by actually constructing and comparing the regions.

Now that we have more insight into the semantics of the regions we can examine how ties in the maximization over previous vectors can further complicate matters. If we find two different previous $\alpha(t-1)$ vectors (call them $\alpha^j(t-1)$ and $\alpha^k(t-1)$) that give the same value in the formula

$$\iota(\pi, a, \theta) = \arg\max_k \Big[\sum_{i,j} \pi_i p_{ij}^a r_{j\theta}^a \alpha_j^k(t-1)\Big],$$

then we have two equally valid and distinct $\iota(\pi, a, \theta)$ functions. As just

[8] $|\mathcal{A}|M^{|\mathcal{A}||\Theta|}$

previously shown, these could represent two completely different regions. So if we chose a belief state π right on the borders of these two regions (assuming for now that they are indeed different), we have two choices for which region we could define. The problem here is that you actually need to consider both and for exactly the same reason that we had to consider both in the previous case of tiesgiven earlier. The same problems with arbitrarily picking one of these apply here; we might miss a region or we might loop around forever repeatedly generating the same π and choosing the same region.

The bookkeeping required to keep all of this straight in this case is an order of magnitude more complex than in the previous case. In the previous case, we could only have as many ties as there were actions, so we could imagine keeping a list of the ties and handling them one after the other. However, in this case there are potential for ties for each possible combination of a and θ. Recall that to define a vector, $\alpha_a(t)$, we must first find an index of a previous vector in $\mathcal{V}^*_{t-1}$ for each observation. As we perform the maximization for each observation, θ, we have the potential of geting as many as $|\mathcal{V}^*_{t-1}|$ ties. Assume that for one observation there are x ties and that there are no ties for any of the other observation.This means that there are potentially x different regions we can choose to examine. If instead another observation yielded y ties, then combined there could be as many as xy regions. Needless to say, the more ties there are, the more complex the process becomes. Implementing the routines to handle this, though possible, is a major headache.

Individually, the two types of ties that can occur in the maximization procedures are bad enough, but they actually interact in an unpleasant way. The previously discussed ties for the $\alpha_a(t)$ vectors assumed that they were all tied while using the same $\iota(\pi, a, \theta)$ function. Now we see that we may end up with bunch of these functions and each one of that bunch has the potential to have ties in the $\alpha_a(t)$ vectors it produces.

All of this might appear to be needless theoretical worry over cases that aren't likely to happen in practice, however just the opposite is true. These problems of ties will occur precisely on the borders between regions. What is the likelihood that we get points on these borders? The nature of the algorithm actually guarantees that we get points on the borders. Recall that all but the first belief point are generated from the LPs, which we have explained can only give us points on the borders of the regions. We would really like the LPs to give us points in neighboring regions, but we can only get points on the borders of neighboring regions. Therefore we spend all of our time directly on the borders and in the corners (vertices) where the ties are liable to happen. Thus, handling ties is a very real problem that must be dealt with when implementing this algorithm.

The last complication we discuss is not nearly as intimidating as the previous problems, but nonetheless, it is a difficulty that must be dealt with. With all these regions being explored and with all the belief states being churned out by the LPs, we are bound to get duplicate belief points to add to the search list. If we have gone through all of the trouble to make sure we have covered every base (handling the ties) for a particular belief state π, then there is no reason to consider this point again. This requires us the keep track of all the π we have dealt with. It is easy enough to keep a list of all the search point, but the major trouble is that we are dealing with floating point comparisons. If we strictly compare the numbers (i.e., literal equality comparisons of double precision numbers), we might be duplicating a lot of effort since what are really the same points could be slightly differnt due to machine rounding errors or errors inherent in the algorithm. However, if we allow too much freedom in deciding which points are the same, we might miss small regions of the value function. As the problem sizes grow larger, the likelihood that there will be these small regions increases.

Sondik's algorithm as described is quite inefficient and in the following sections we present three ways that it could be made less wasteful. The first two of these are due to Sondik and used in his own implementation.

4.6.4 Neighbor Optimization

The geometric interpretation of the value function and the regions they impose upon the belief states allow us to exploit the geometric properties of these regions. When we define a region and then look for the neighboring regions, we are finding each regions' adjacent neighbors. If we store this neighbor information, we can exploit it to restrict the number of constraints in the linear programs on the next time step. Recall the set of constraints:

$$\sum_{i,j} \pi_i p_{ij}^a r_{j\theta}^a \left(\alpha^k(t-1) - \alpha^{\iota(\pi,a,\theta)}(t-1) \right) \leq 0, \qquad \forall \theta, k \neq \iota(\pi, a, \theta).$$

This says that we must construct a constraint for every previous vector $\alpha^k(t-1)$, but actually we only need to consider the $\alpha^k(t-1)$ that are neighbors to $\alpha^{\iota(\pi,a,\theta)}(t-1)$. If we kept this region adjacency information from the previous iteration, we know exactly which $\alpha^k(t-1)$ are truly constraining the current $\alpha^{\iota(\pi,a,\theta)}(t-1)$ and which ones definitely cannot be constraining it.

4.6.5 LP Optimization

Even with the above optimization, the number of constraints in the linear programs will be large. The following insight is directly from [16]:

> The procedure can be made more efficient if, for each iteration of the linear programming problem with the k^{th} inequality as the objective function, all other constraints are tested as objective functions to see if they are optimized at the current feasible solution. If a constraint is optimized at any point, then either this constraint forms a boundary of the region (a zero slack variable) or is a superfluous constraint (a nonzero slack variable). Once a constraint has been optimized, it need not be used as the objective function. We have found that this procedure typically decreases the number of linear programming iterations by approximately 50 percent.

We have no experience with this optimization and therefore have offered it in their own words. The geometric intuition behind this method is that as an LP solution progresses, it moves from corner point to corner point of the constrained region. Each corner point lies on one of the boundaries of the region and all this technique does is to see which border point it is on at each iteration by substituting each possible border (i.e., constraint). However, this requires an intimate relationship with the LP package. Unless you have written the code specifically for this application, it is unlikely to be easy and may be impossible for you to get this functionality out of an LP package.

4.6.6 Dominated Constraints

This is the simplest of all the optimizations and just requires that we compare each constraint, as it is generated, with all the others to determine whether it is component-wise dominated by some other constraint. Since our variables of the LP (π_is) are constrained to be non-negative, this simple scheme works.

Whether or not this checking is worth the trouble depends on you particular implementation and the speed of the LP package being used. Some LP packages will implicitly perform this check for you as part of its optimization scheme.

4.7 Cheng's Algorithms

In [4] two new algorithms are presented. They are both heavily influence by Sondik's One Pass algorithm, but typically require less computation time. One major deviation present in Cheng's algorithms is the elimination of the use of linear programming. Instead, Cheng opts for the interior point method for convex polytopes (see [10] and [11]) to find the corner points of the regions define on the belief space. This interior point method is similiar to linear programming, but can require significantly more time. The reason for this is that the interior point method will search for and enumerate every possible vertex (corner point) in the region, whereas linear programming will search for a single point only. The reasons for this change from linear programming are discussed shortly.

4.7.1 Cheng's Relaxed Region Algorithm

This algorithm is almost exactly the same as Sondik's One Pass algorithm except that each region is specified with fewer constraints. In each step of Sondik's algorithm, a region for the newly discovered $\alpha(t)$ vectors of the value function, $V_t^*(\cdot)$, is defined which is sure to be no bigger than the region the $\alpha(t)$ vector actually defines over the belief space. This is the conservative approach and thus, the actual regions for each of the true vectors of the value function will have to be built up out of a bunch of these smaller regions that Sondik defines. Cheng reverses this approach and defines regions that will typically be larger than the actual vectors' regions (i.e., relaxed regions).

The algorithm can proceed exactly the same as Sondik's with two minor, but important changes. The first occurs in the set of constraints and the other is in the use of the interior point method instead of normal linear programming. Recall how we need to keep a search list of belief points and that, for each belief point, we will find the true vector $\alpha^*(t)$, its associated action a^* and all the other $\alpha_a(t)$ vectors that were the losers in the maximization over actions. We also have the set of all the previous $\alpha(t-1)$ vectors. The only modification necessary is in the construction of the regions or the set of constraints for the LP. The following is the set of constraints for the relaxed regions of Cheng:

$$\sum_i \pi_i \left(\alpha_{a,i}(t) - \alpha_i^*(t)\right) \leq 0, \qquad \forall a \neq a^*$$

$$\sum_{i,j} \pi_i p_{ij}^{a^*} r_{j\theta}^{a^*} \left(\alpha^k(t-1) - \alpha^{\iota(\pi,a^*,\theta)}(t-1)\right) \leq 0, \qquad \forall \theta, k \neq \iota(\pi, a^*, \theta)$$

$$\sum_i \pi_i = 1$$

$$\pi_i \geq 0, \qquad \forall i.$$

If you compare this set of constraints with Sondik's it might take a while to even notice that they are different. The only change is the second constraint which is no longer defined over all actions. This constraint now only applies for each θ and previous $\alpha^k(t-1)$ vector with the action being fixed as the action associated with $\alpha^*(t)$.

In the interior point method, unlike and LP, there is no objective function. The constraints serve to define a region and the interior point algorithm will systematically visit each vertex (corner point) of this region. By using an interior point algorithm to ensure that every vertex of one of these relaxed regions is discovered, Cheng can guarantee that all of the optimal value function vectors will eventually be discovered in a finite number of steps, despite the construction of regions that are larger than they actually should be.

Before we discuss how generating all the optimal vectors can be guaranteed, we should point out the crucial difference between utilizing an interior point method and a linear programming method. If you recall from the discussion of Sondik, the linear programs will typically have an infinite number of potential points they can return and, by the nature of the LP method (usually simplex), we will get only one of the corner points. This works fine for Sondik's method as was previously discussed, but can lead to incorrect results if combined with the relaxed region algorithm.

For example, in Figure 31 we see the two regions defined by $\alpha^*(t)$ (one for each algorithm) which were generated for belief state π; Sondik's region is bounded by $ABCD$ and the relaxed region of Cheng by ABE. Let us assume that the true region of the value function for $\alpha^*(t)$ is $ABCD$ (which just so

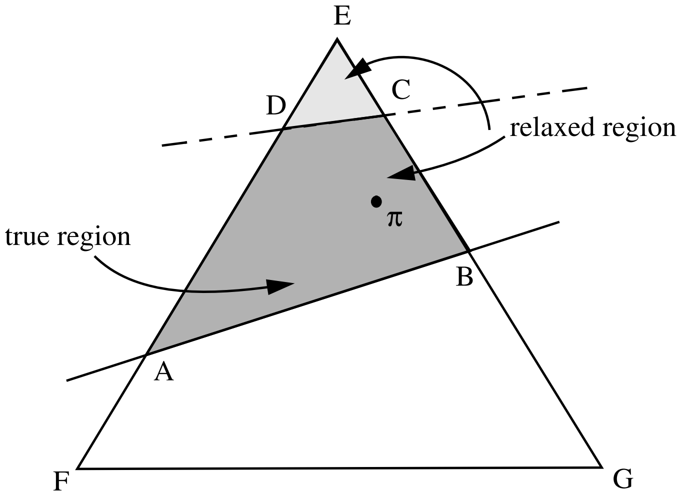

Figure 31: Points and relaxed regions

happens to be the one found by Sondik's method). Sondik's algorithm will show two neighboring regions. For neighboring region CDE the LP method will return either point C or D and for neighboring region $ABGF$ either A or B. Even though we do not know exactly which point the LP might return, either one is fine since they will both lead to the other regions. In contrast, look at the relaxed region ABE, this one looks to only have one neighboring region and so, if we employ the LP method, the LP will return either point A or B. This is not sufficient, because we will never choose a point in the region DCE, which is a different region of the optimal value function that must be found. However, if we always make sure we find each corner point of the region, then we are fine, since the point E will lead to the region CDE. This illustrates why Cheng could not use normal linear programming in conjunction with his relaxed regions.

Cheng demonstrates that by employing the interior point method, his algorithm will always uncover all the regions. Furthermore, since his regions are larger than those that Sondik provides, it should execute faster. This algorithm only needs to find a single relaxed region for each actual vector in the optimal value function, which can be substantially fewer than the number defined by Sondik's One Pass method. Cheng has experimental results that verify that this is actually the case. Cheng's work is also explicit about handling the case of ties, whereas Sondik's work lacked discussion about

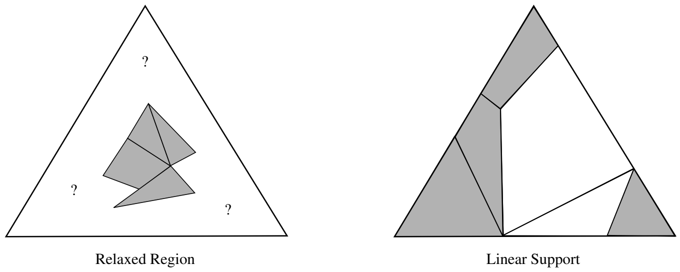

Figure 32: Comparison of Cheng's two algorithms

these cases.

4.7.2 Cheng's Linear Support Algorithm

Cheng proposes another algorithm which employs the same idea of defining larger regions. However, this Linear Support algorithm has the advantage that it can be used to find near optimal answers as well. With the previous Relaxed Region algorithm, we start at a point and move from one region to another, slowly exploring the space of beliefs. The Linear Support algorithm, on the other hand, defines an approximate value function over the entire belief space and slowly works to refine this approximation until it reaches the optimal value function or until the approximation is within the desired tolerance from the optimal.

The differences between Cheng's two algorithms is fairly subtle, but the main difference is illustrated in Figure 32. Here we have stopped both algorithms in the middle of computing the value function, $V_t^*(\cdot)$. The figures illustrate the regions induced on the belief simplex of the vectors thus far discovered. At this point the Relaxed Region algorithm has the exact value function for the shaded region, however nothing is know about the value function in the un-shaded regions. In contrast, the Linear Support algorithm has a value function defined over the entire belief simplex. The shaded regions show where the current approximation is actually equal to the optimal value function. The value function over the unshaded regions are known to be within some factor of the optimal value function.

The name "Linear Support" comes from Cheng's terminology for the linear segments of the value function which we have been calling $\alpha(t)$ vectors. In Cheng's thesis he refers to these vectors as the linear supports of the value

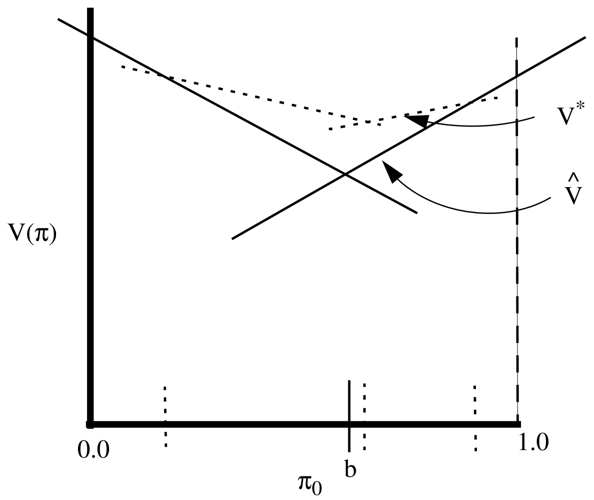

Figure 33: First step of Linear Support algorithm

function.

The algorithm starts by initializing a search list with the extreme points on the belief simplex (e.g., [1, 0, 0, ...], [0, 1, 0, 0, ...], [0, 0, 1, 0, ...], etc.)and an empty set of vectors, $\hat{\mathcal{V}}$ that at any point in time form the current approximation. For each of these points the true $\alpha(t)$ vector is calculated and added to $\hat{\mathcal{V}}$ (calculated from the usual recursive value function formula.) These vectors now form an approximation to the value function which we will call $\hat{V}_t(\cdot)$ and the vectors induce a partition on the belief space, since for each point of the belief space, one of these vectors must give the maximum value when compared to the rest. For example, Figure 33 shows the optimal value function as a dotted line and the approximation generated from the corner points as solid lines. Notice that close to the extreme points of the belief simplex the approximation is exact. This is for a case where $|\mathcal{S}| = 2$ and it shows the belief space partition generated along the horizontal axis. This figure shows an example where there are four linear regions that comprise the optimal value function. These four linear segments impose a partition on the underlying belief states as shown with the dotted lines across the π_0 axis. When we only calculate the linear segments for the extreme points of the simplex, we get parts of the optimal value function (near the extreme points), but we also under-estimate the optimal value function at many of the interior points since we have not yet found the true vectors for these points.

The key insight behind this algorithm, is that no matter what the optimal value function is, the largest difference between the approximation, $\hat{V}_t(\cdot)$, and the optimal value function, $V_t^*(\cdot)$ will always occur at one of the corner points of the approximation. This follows from the properties of piecewise linear functions. Notice that the optimal value function must always lie equal to or above the approximation. Since both the true function and the approximation are piecewise linear and convex the largest difference must occur at a corner point. This is proved in [4].

With this handy piece of information, Cheng then finds all the corner points of the regions in the partition induced by the approximation. Here again, Cheng utilizes an interior point algorithm instead of linear programming, since he must ensure all corner points are generated. For Figure 33, this will consist of three points, the two edges of the simplex and one interior point, b. Since we have already handled the simplex corner points we only need to consider the single interior point, b. This generalizes so that of all the corner points found from all the regions, we disregard those we have seen before and add those we have not seen before to the search list. Note that this requires you to keep track of all points you have used previously, just as in Sondik's One Pass algorithm.

The next step is to pick a point out of this search list and find the true $\alpha(t)$ vector (support) for that point. If this vector is different from all the other ones in our current approximation, then we add it to $\hat{\mathcal{V}}$ to arrive at a new approximation, $\hat{V}_t(\cdot)$. Figure 34 shows the succeeding approximation for the example shown previously, after the vector for point b has been added to $\hat{\mathcal{V}}_t$.

We can now repeat this whole procedure with the new approximation: find corners of region; add new point to search list; generate $\alpha(t)$ vectors; add new vectors to $\hat{\mathcal{V}}$ to get a new approximation, $\hat{V}_t(\cdot)$. You can see in Figure 34 that calculating the new vector at the new corner point, c, will lead to the only remaining undiscovered vector.

Although this algorithm will work, as described, it is a bit wasteful. Each successive approximation imposes a new partition on the belief space. At each iteration we do not need to find all the corners of all the regions of the current partition. Cheng shows that if we just find the corner points for the region of the newly added vector we can get the same result as before with much less computation. This also guarantees that the algorithm will

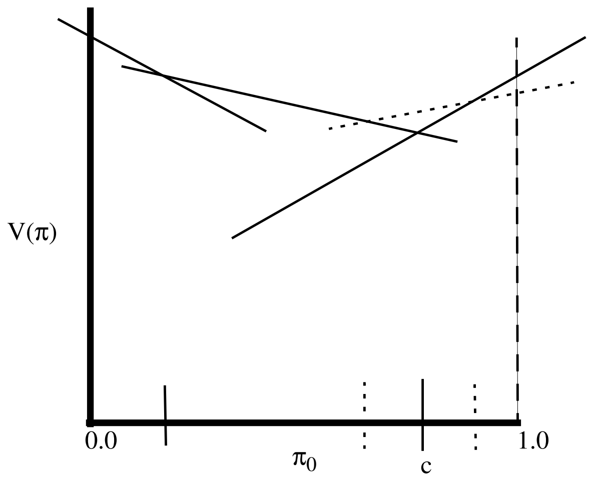

Figure 34: Second step of Linear Support algorithm

only examines one region for each true vector (support) in the optimal value function. This is its major advantage over Sondik's method which typically uncovers more regions than are actually imposed by the vectors in the optimal value function.

Linear Support is similar to the Relaxed Region algorithm because, as Figures 33 and 34 show, the region over the belief space in an approximation tends to be larger than it ultimately will be in the optimal value function. There is another point Cheng makes about uncovering the interior points of the regions which states that you only need to find corner points for all but one of the regions. This follows because the corner points of any single region imposed by a piecewise linear convex function must also be corner points of some other region. Figure 35 shows that if we generate all the corner points for the non-shaded regions, we will also have generated all the corner points of the shaded region.

Cheng employs the Linear Support algorithm as an approximation scheme as well. To do this he merely calculates the actual difference between the optimal value function and the approximation at the vertex points discovered for each region defined by the approximation, $\hat{V}_t(\cdot)$. By taking the largest difference in the two values he gets a bound on the current approximation. Remember, the largest difference must occur at one of these vertices. He can stop his algorithm any time this maximum error difference is within some

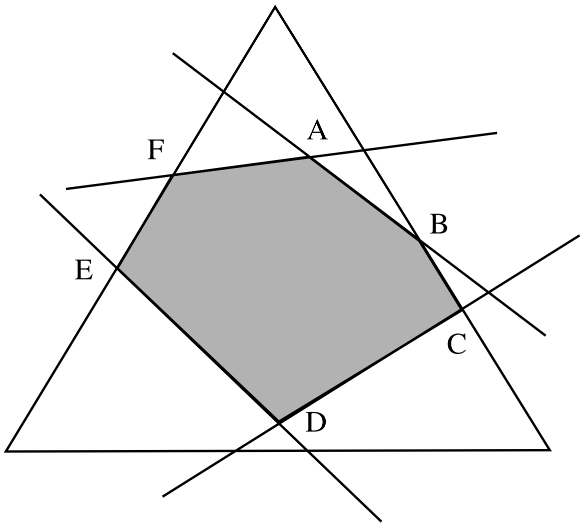

Figure 35: Example of why one region doesn't need to have its corners checked

tolerable range.

The final difference between the Relaxed Region algorithm and Linear Support is in the manner the regions are defined. Recall that the Relaxed Region algorithm used constraints based mostly upon previous $\alpha(t-1)$ vectors. Here in the Linear Support algorithm, only the vectors currently in the approximation, $\hat{\mathcal{V}}_t$, are utilized. The goal is to determine if, for each vector in the approximation, the region it defines is the same as the region it would define in the optimal value function. To accomplish this, we examine the values of the approximation at the vertices of the region and compare them to the values of the optimal value function at those points. If at all points the approximation, $\hat{V}_t(\cdot) = V_t^*(\cdot)$ then we know that there are no points within this region that differ from the optimal, since the optimal function is piecewise linear.

The region constraints are quite simple for this algorithm:

$$\sum_i \pi_i \left(\hat{\alpha}_i(t) - \alpha_i^k(t)\right) \leq 0, \quad \forall \alpha_i^k(t) \in \hat{\mathcal{V}}_t$$

$$\sum_i \pi_i = 1$$

Here $\hat{\alpha}(t)$ is the vector in $\hat{\mathcal{V}}_t$ whose region we wish to examine. The Linear Support algorithm is summarized as follows:

```
(1) Initialize V̂_t with all vectors generated from
  extreme points of belief simplex and mark each vector.
(2) Select a marked vector, α̂(t), from V̂_t.
  If there are none, then V̂_t = V*_t.
(3) Otherwise, set up region for α̂(t) and find all
  vertices of the region.
(4) For each vertex:
  (i) If vertex has already been used, then ignore it,
    otherwise construct vector, α'(t), for this point.
    and mark it.
  (ii) if α'(t) ∈ V̂_t, then ignore it,
    otherwise add α'(t) to V̂_t and mark it.
(5) Unmark α̂(t).
(6) Go to step (2).
```

4.8 The Witness Algorithm

There is a shortcoming of Cheng's Linear Support algorithm which this next algorithm addresses. Cheng's algorithm must find every vertex of each region it defines. Each of these regions is a convex polytope and as such can have an exponential number of vertices. The number of vertices is exponential in the number of faces in the polytope and the number of faces is dictated by either the number of vectors in the current approximation, up to $|\mathcal{V}_t^*|$, or the number of variables in the state space, $|\mathcal{S}|$.

The Witness algorithm was derived directly from Cheng's Linear Support algorithm and employs much of the same machinery. There are two important differences in the Witness algorithm. First, it does not deal directly with an approximation to the value function. Instead, it defines related value functions and then builds these function up from approximations much the same as Cheng builds up $\mathcal{V}_t^*$ from successive approximations, $\hat{\mathcal{V}}_t$. These related values functions, once found, can be used to directly construct $\mathcal{V}_t^*$. The other major deviation from Cheng is in the way points are found to refine the approximation. Note that although it uses slightly different value functions, the method of successive approximations is still the same.

Like all other algorithms we start the algorithm with a set of vectors, $\mathcal{V}_{t-1}^* = \{\alpha^0(t-1), \alpha^1(t-1), \ldots, \alpha^{M-1}(t-1)\}$. This set is the piecewise linear representation for the optimal value function at the $(t-1)^{st}$ time step where

$$V_{t-1}^*(\pi) = \max_k \pi \cdot \alpha^k(t-1).$$

The output of the algorithm will be a set of vectors $\mathcal{V}_t = \{\alpha^0(t), \alpha^1(t), \ldots, \alpha^{N-1}(t)\}$ representing the optimal value function, $V_t^*(\cdot)$, for the t^{th} time step.

Unlike the other algorithms, here we need to define an alternate set of vectors, $\mathcal{Q}_t^a = \{\alpha_a^0(t), \alpha_a^1(t), \ldots, \alpha_a^{N-1_a}(t)\}$. This set of vectors represent the piecewise linear action-value function, $Q_t^a(\cdot)$, for performing action a at time t and performing optimally thereafter. The $V_{t-1}^*(\cdot)$ value function gives us the value for performing optimally thereafter. We will need a separate $\mathcal{Q}_t^a$ set of vectors for each action, a. This value function can be specified in terms of immediate rewards and previous $\alpha^k(t-1) \in \mathcal{V}_{t-1}^*$ vectors as

$$Q_t^a(\pi) = \sum_i \pi_i q_i^a + \gamma \sum_{i,j,\theta} \pi_i p_{ij}^a r_{j\theta}^a \alpha_j^{\iota(\pi,a,\theta)}(t-1).$$

Here, as in Sondik's One-Pass algorithm

$$\iota(\pi, a, \theta) = \arg\max_k \sum_{i,j} \pi_i p_{ij}^a r_{j\theta}^a \alpha_j^k(t-1).$$

With these formulae we can express component i of a vector in $\mathcal{Q}_t^a$ for a given point, π, by

$$\alpha_{a,i}^k(t) = q_i^a + \gamma \sum_{j,\theta} p_{ij}^a r_{j\theta}^a \alpha_j^{\iota(\pi,a,\theta)}(t-1). \tag{10}$$

This is exactly analogous to generating a vector for $\mathcal{V}_t^*$ given a particular belief point except we do not need to maximize over the actions since a is fixed for each set $\mathcal{Q}_t^a$.

The Witness algorithm will first construct the $\mathcal{Q}_t^a$ sets and then construct the desired $\mathcal{V}_t^*$ set of vectors from these. In constructing a particular set, $\mathcal{Q}_t^a$, we will incrementally build up to the full set by successive approximations, $\hat{\mathcal{Q}}_t^a$. The approximations are such that at all times $\hat{\mathcal{Q}}_t^a \subseteq \mathcal{Q}_t^a$. Note also that

$$\mathcal{V}_t^* \subseteq \bigcup_a \mathcal{Q}_t^a,$$

since the value function, $V_t^*(\cdot)$, is the same at a belief point as the $Q_t^a(\cdot)$ function when the action a is optimal at that point. Since we have accounted for all possible actions, no matter what the optimal action for a point is, the corresponding vector will be in $\bigcup_a \mathcal{Q}_t^a$.

4.8.1 Constructing $\mathcal{Q}_t^a$

In the following discussion it will prove useful to define the following for a particular vector $\hat{\alpha}_a^k(t) \in \mathcal{Q}_t^a$:

$$region(\hat{\alpha}_a^k(t)) = \{\pi | \pi\hat{\alpha}_a^k(t) \geq \pi\hat{\alpha}_a^l(t), \forall \hat{\alpha}_a^l(t) \in \hat{\mathcal{Q}}_t^a\}.$$

This is just a convenient way to define a region of the belief simplex over which a particular vector gives the best value, given some set of vectors $\hat{\mathcal{Q}}_t^a$.

For each a we will need to construct a set of vectors, $\mathcal{Q}_t^a$, and the construction of each is identical. We begin with an empty set $\hat{\mathcal{Q}}_t^a$. Initially, we choose any belief point and construct a vector using formula 10 and add it to $\hat{\mathcal{Q}}_t^a\}$.

The problem now becomes; given an approximation, $\hat{\mathcal{Q}}_t^a$, determine if there exists a belief state, π^*, such that the following conditions hold:

$$\pi^* \cdot \alpha^*(t) > \pi^* \cdot \hat{\alpha}_a(t)$$

$$\pi^* \in region(\hat{\alpha}_a(t))$$

$$\alpha^*(t) \in \mathcal{Q}_t^a \;\; and \;\; \alpha^*(t) \ni \hat{\mathcal{Q}}_t^a$$

$$\hat{\alpha}_a(t) \in \hat{\mathcal{Q}}_t^a.$$

In words, it says to look at the belief states for which a particular vector in the approximation, $\hat{\alpha}_a(t)$, provides a maximum value. If any of these belief states have the property that there is a vector in the true $\mathcal{Q}_t^a$ set (and not already in $\hat{\mathcal{Q}}_t^a$) that gives a larger value, then we have found a new vector that we can add to and refine our approximation $\hat{\mathcal{Q}}_t^a$.

Iterating this for each vector, $\hat{\alpha}_a(n)$, in the approximation will eventually lead us to the desired set of vectors $\mathcal{Q}_t^a$. However, there are a few subtleties associated with this iteration procedure. First, we will choose a vector in the approximation, $\hat{\alpha}_a(n) \in \hat{\mathcal{Q}}_t^a$, and determine if there is a belief state, π^*, that satisfies the conditions above. If we do not find one, then we are done with this vector and never need to consider it again. However, if we do find a belief state, we cannot eliminate this vector from those we are iterating over, since it is possible for one vector to give rise to a number of belief points. This is in constrast with Cheng's Linear Support algorithm. Cheng finds every single vertex for a vector's region and thus, he can immediately know never to examine this vector again. Since the Witness algorithm does not exhaustively go through all the vertices of the region, it can never be sure we are finished with this vector until it no longer gives us useful belief points. This will be made clearer in the discussion to follow. The other subtlety is that once a belief point is found, byt the Witness algorithm, and a new vector for this point generated, it too must be added to the list of vectors we are iterating over.

Up until now we have been vague about how to actually find such a belief state for a given a vector in the approximation. To find if this belief state exists, we start with the observations that

$$\hat{\alpha}_{a,i}(t) = q_i^a + \gamma \sum_{j\theta} p_{ij}^a r_{j\theta}^a \alpha_j^{\iota(\pi,a,\theta)}(t-1)$$

and

$$\alpha_i^*(t) = q_i^a + \gamma \sum_{j\theta} p_{ij}^a r_{j\theta}^a \alpha_j^{\iota(\pi^*,a,\theta)}(t-1).$$

Here $\hat{\alpha}_a(t)$ is the vector chosen from $\hat{\mathcal{Q}}_t^a$ and $\alpha^*(t)$ is the new vector that we want to find (assuming that it exists). Notice that the first equation differs from the second only in the belief state, π, since it is not necessarily equal to π^* of the second equation. The belief state, π, is the one we used when we initially generated the vector $\hat{\alpha}_a(t)$. The belief state, π^*, is the one we are trying to find (or to show that it cannot exist). It might seem as if we need to remember the belief state that we used to determine $\hat{\alpha}_a(t)$, but we do not. However, we will need to have remembered the indices of the vectors that the $\iota(\cdot,\cdot,\cdot)$ function generated for that belief state and for every observation.

Therefore, every time we generate a vector from a belief point we will need to store $|\Theta|$ values to represent the previous vectors' indices used in the inner summation of formula 10. We will store the vector indices used for each vector in $\hat{\mathcal{Q}}_t^a$ and notate the ones for $\hat{\alpha}_a(t)$ as $\hat{\imath}(\theta)$.

Now, if there exists a π^* as described before, then

$$\pi^* \cdot \alpha^*(t) > \pi^* \cdot \hat{\alpha}_a(t),$$

or in long form, with $\hat{\iota}(\theta)$ being substituted for $\iota(\pi, a, \theta)$:

$$\sum_i \pi_i^* \left(q_i^a + \gamma \sum_{j,\theta} p_{ij}^a r_{j\theta}^a \alpha_j^{\iota(\pi^*,a,\theta)}(t-1) \right) > \sum_i \pi_i^* \left(q_i^a + \gamma \sum_{j,\theta} p_{ij}^a r_{j\theta}^a \alpha_j^{\hat{\iota}(\theta)}(t-1) \right).$$

Since the first terms, $\sum_i \pi_i^* q_i^a$, are identical for both sides, we can simplify[9] it to

$$\gamma \sum_\theta \sum_{i,j} \pi_i^* p_{ij}^a r_{j\theta}^a \alpha_j^{\iota(\pi^*,a,\theta)}(t-1) > \gamma \sum_\theta \sum_{i,j} \pi_i^* p_{ij}^a r_{j\theta}^a \alpha_j^{\hat{\iota}(\theta)}(t-1).$$

Notice that the discount factor, γ is now extraneous since we can divide both sides of the equation by γ. So it must be the case that at least one of the terms in the summation over observations, $\sum_\theta$, on the left is strictly larger than the corresponding term in the formula for the $\hat{\alpha}_a(t)$ vector on the right. In other words, since the only difference between the two sides is $\iota(\pi^*, a, \theta)$ and $\hat{\iota}(\theta)$, there must be another setting of indices for the $\hat{\iota}(\theta)$ function that yields a better value at the belief state π^*, i.e.,

$$\exists \theta, k, \ s.t. \ \sum_{i,j} \pi_i^* p_{ij}^a r_{j\theta}^a \alpha_j^k(t-1) > \sum_{i,j} \pi_i^* p_{ij}^a r_{j\theta}^a \alpha_j^{\hat{\iota}(\theta)}(t-1).$$

If we can find such a π^*, θ and k where

$$\pi^* \in region\left(\hat{\alpha}_a(t)\right),$$

then we have found a "witness" point indicating that our current approximation, $\hat{\mathcal{Q}}_t^a$, can be further refined by including the vector constructed at this point π^*.

What the Witness algorithm does is to examine each vector in the current approximation to see if for each combination of $\alpha^k(t-1)$ and θ, we can find a π^*. If a point is found, then we have a point where our approximation differs from the actual $\mathcal{Q}_t^a$. The algorithm then constructs a vector for this point and adds it to $\hat{\mathcal{Q}}_t^a$. If such a point does not exist, then we never need to consider this combination of $\alpha^k(t-1)$ and θ again for the chosen vector again. Additionally, once all combinations of $\alpha^k(t-1)$ and θ have been eliminated for a vector $\hat{\alpha}_a^k(n) \in \hat{\mathcal{Q}}_t^a$, we never need to consider this vector again for finding belief points.

[9] We have separated the $\sum_\theta$ in preparation for the next step.

An important point to remember is that when a π^* is found, we must try that combination of $\alpha^k(t-1)$ and θ again, since it is possible for this combination to give rise to more than one belief point.

The algorithm for constructing $\mathcal{Q}_t^a$ is summarized as follows:

```
(1) Initialize Q̂_t^a with a vector generated from
  any belief point and mark it.
(2) Select a marked vector, α̂_a(t) from Q̂_t^a.
  If there are none, then we are done and Q̂_t^a = Q_t^a.
(3) For each possible combination of α^k(t-1) ∈ V*_{t-1} and
  θ ∈ Θ:
  (i) Construct and solve an LP with α̂_a(t), α^k(t-1)
    and θ (LP is shown below).
  (ii) If δ > 0, then construct a vector from solution point
    using formula 10, add it to Q̂_t^a,
    mark it, and repeat (i) for this same combination.
  (iii) If δ ≤ 0, then just move on the next combination of
    α^k(t-1) and θ.
(3) Unmark α̂_a(t).
(4) Go to (2).
```

Each LP will have the following form:

$$max : \delta$$

$$s.t. \ \sum_i \pi_i \hat{\alpha}_{a,i}(t) \geq \sum_i \pi_i \alpha^l_{a,i}(t), \ \ \forall \alpha^l_a(t) \in \hat{\mathcal{Q}}^a_t$$

$$\sum_{i,j} \pi_i p^a_{ij} r^a_{j\theta} \left(\alpha^k(t-1) - \alpha^{\hat{\imath}(\theta)}(t-1) \right) \geq \delta$$

$$\sum_i \pi_i = 1$$

$$\pi_i \geq 0, \ \ \forall i.$$

Notice that the same extra variable trick, δ, is employed as was used in the Monahan LPs to utilize strict inequalities in the LP framework. Therefore, we are then only concerned with solution points when $\delta > 0$.

4.8.2 Constructing $\mathcal{V}_t^*$

Once we have constructed all of the sets $\mathcal{Q}_t^a$ for each action a, we can then combine them to form the set $\mathcal{V}_t^*$ that represents the optimal value function, $V_t^*(\cdot)$ which is what we are really interested in. To do this we employ the reduction phase of Monahan's algorithm. Recall that Monahan's algorithm consisted of two phases. The first phase enumerated many vectors to form the set $\mathcal{V}_n^+$, which was a super set of the desired set, $\mathcal{V}_n^*$. The next step, the reduction phase, eliminated the useless vector from $\mathcal{V}_n^+$. If we combine all of the $\mathcal{Q}_t^a$ sets to get a single large set, we know that it is a superset of $\mathcal{V}_t$. Therefore we treat $\bigcup_a \mathcal{Q}_t^a$ as the set $\mathcal{V}_n^+$ and use Monahan's reduction method to obtain $\mathcal{V}_t^*$.

While this may seem to be a lot more work than Monahan's algorithm, since we need Monahan's as a subroutine, in practice there are significantly fewer vectors in $\bigcup_a \mathcal{Q}_t^a$. than are considered in Monahan's algorithm. Furthermore, there is significantly less work in creating the intermediate $\mathcal{Q}_t^a$ sets than there is in enumerating all possible vectors.

4.8.3 Dealing with Ties

The one complication that must be handled is the case when there are more than one possible vector for a given point. This case arises when generating a vector from a belief point where there are ties in either of the two possible maximization procedures; over actions, formula 7, or over previous vectors, 6.

The actual method that needs to be used to break ties among vectors is conceptually straightforward. Unfortunately, the proof that this procedure is correct is quite tedious. Here we will only describe the procedure, though [8] gives the formal proof in its entirety.

As mentioned before, there are two places where a maximization process can result in ties. The simplest to address is when ties result from formula 7. In this case we merely list all the vectors that result in the same maximal value for the given belief state and choose the lexicographic winner. The vector that wins is the vector (and action) that should be used.

The lexicographic winner is chosen by examining the first component of the vector. If there is a clear-cut winner then it is the one chosen. Otherwise, if there are ties in that component, then we examine the second component for all of those that are tied on the first component. The same procedure is repeated until a unique lexicographic winner is determined.

Formula 6 can also result in ties and in this case, the exact same lexicographic ordering can be employed. The index of lexicographic winner is the one that should be chosen for use in the sum over θ from formula 7.

4.9 Algorithm Analysis

Littered throughout the discussion of the algorithms are references to the worst case running times for each of the algorithms. In this section we will focus our attention on this topic describing in more detail how these worst case running times arise. We will progress through the algorithms in the same order in which we discussed them previously. All of the analysis deals with the complexity of producing one set of vectors from the previous set. The actual complexity of each algorithm on a complete k-horizon problem is actually much more complicated to analyze since the complexity of each iteration depends upon the complexity of the previous one. We can say that in the worst case, the general k-horizon POMDP problem can have as many as

$$\sum_{1 \leq t \leq k} |\mathcal{A}|^{\frac{|\Theta|^{t+1}-1}{|\Theta|-1}}$$

vectors. Each term in the summation represents the worst case number of vector for each iteration and a k-horizon solution requires all the intermediate results since the policy is non-stationary. The terms of the summation are the total number of possible policy trees for each time step as discussed earlier. If we could construct a diabolical problem where all of these trees were useful for some belief state, then any algorithm we could possibly derive would have to, at a minimum, generate this many vectors, since each vector represents a useful policy tree beginning at the k^{th} time step.

Before we begin the analysis we will just review some notation; $M = |\mathcal{V}_{t-1}^*|$ is the minimum number of vectors that are required exactly specify the value function, $V_{t-1}^*(\cdot)$ for the previous time step and $N = |\mathcal{V}_t^*|$ is the same except for the current time step's value function, $V_t^*(\cdot)$. We will use M and N to simplify the appearance of many of the ensuing formulas.

4.9.1 Monahan's Algorithm

The first step of Monahan's algorithm is to generate all the possible vectors that could be formed from the value function (formula 7). As mentioned there are $|\mathcal{A}|M^{|\Theta|}$ vectors that will be generated. The $M^{|\Theta|}$ term is the number of ways we could fill in the M previous vectors in the summation where we pick one for each θ. For each of these summations there are $|\mathcal{A}|$ possible actions that can be used to generate a vector. The algorithm will always require generating this many, this is not a worst case analysis.

The next step in Monahan's algorithm is to construct and solve a linear program for each vector to determine whether or not it is actually useful (i.e., part of the optimal value function). The size of these LPs will vary over time since it can eliminate useless vectors as it proceeds. Nevertheless, the first LP will have $|\mathcal{S}|$ variables and $|\mathcal{A}|M^{|\Theta|}+1$ constraints. For ease of discussion we will let $X = |\mathcal{A}|M^{|\Theta|}+1$ and note that $N \leq X$. There will be a constraint for each of the generated vectors plus one for the simplex constraint, $\sum_i \pi_i = 1$. Since each vector must be evaluated in an LP, this algorithm must solve X linear programs. For each LP if we discover that the vector is part of the optimal value function then we will need to include this as a constraint in all the subsequent LPs. However, if there is no belief state for which this vector can be optimal we can discard it and it will no longer generate a constraint in subsequent LPs. Although we know we will have to specify X LPs, we cannot say exactly how big (i.e., number of constraints) each one will be because it will depend on the order in which we choose to evaluate the vectors.

The best case is when we try all the useless vectors first. This trims down the number of LP constraints as fast as possible where there is one fewer constraint on each subsequent LP. The trimming will eventually stop when we are left will all of the real vectors, but we still need to try them all since we have no *a priori* knowledge of how many there are. These last N LPs will then all have $N+1$ constraints. One way we could look at the best and worst case complexity of the LP solving process is to count the total number of constraints generated. Then, in the best case, the total number of constraints is

$$\sum_{i=N+2}^{X+1} i + \sum_{i=1}^{N}(N+1),$$

which simplifies to

$$\frac{X^2 + 3X - 3N^2 - 5N}{2}.$$

In the worst case we will check all the N true vectors first and the number of constraints will not change until we start trying the useless vectors. The total number of constraints in the X LPs in the worst case is

$$\sum_{i=1}^{N}(X+1) + \sum_{i=N+1}^{X+1} i,$$

which has the close form

$$\frac{X^2 + 3X + 2NX - N^2 + N + 2}{2}.$$

In both the best and worst case, the overwhelming factor is the X^2 term which shows that for large problems the dominating factor will be $M^{|\Theta|}$.

4.9.2 Eagle's Variant of Monahan's Algorithm

Eagle's variant of Monahan's algorithm tries to reduce both the number of LPs that have to be solved and the number of constraints in each. Unfortunately, it becomes very hard to analyze the savings since it depends on the specific problem. This algorithm relies on being able to eliminate vectors that are component-wise dominated by previously or subsequently generated vectors. The number of such vectors that will exist cannot be determined in advance since it will depend on the specific parameters of the POMDP formulation. Notice that even with this version all the X vectors must still be generated.

Let us assume that when we generate all of the vectors there will be Y of them that are component-wise dominated by others. Right away we get savings in the linear programming step because we now only have $X - Y$ vectors. We can substitute $X - Y$ for X in the previously shown formulas to get new measures for the complexity in this case. However, we also pay a computation price in the comparisons that needed to be done to check for the domination. The actual complexity that is added will depend on the specific way the domination is checked. There are many clever standard search and data structure tricks from the computer science literature that could be used to make this fairly efficient, but the cost must still be factored in.

The last thing to mention about Eagle's optimization is that there do not have to exist any dominated vectors at all. Dominated vectors are useless vectors, but the converse is not true.

4.9.3 Sondik's One Pass Algorithm

By far, the portion of Sondik's algorithm that consumes the most time is the linear programming step. Maintaining the search list of belief states and generating vectors from a specific belief state are trivial computations compared to the vast about of time that must be spent on the large number of LPs this algorithm generates and solves. Therefore we restrict the complexity analysis to the number and size of the LPs that need to be solved.

As discussed, the belief space regions Sondik's algorithm generates are smaller than the true regions in the resulting value function. Since this algorithm sets up an LP for each of these regions, the number of regions will be the deciding factor. Recall that a region is defined so that all of the $\alpha_a(t)$ and $\alpha^*(t)$ remain unchanged. For this to occur, the index function $\iota(\cdot)$ must remain unchanged. The worst case for this algorithm is when it generates a region for every possible instantiation of an $\iota(\cdot)$ function for each possible action. The function can be specified as an $|\mathcal{A}|$ by $|\Theta|$ table where each entry in the table can be one of M possible choices (the previous vector indices). Thus, there are

$$M^{|\mathcal{A}||\Theta|}$$

possible $\iota(\cdot)$ functions. However, for each of these functions there may be areas in the belief space where different actions are optimal (i.e., the index function is unchanged, but a different $\alpha_a(t)$ is larger than the rest).

So the number of possible regions is

$$|\mathcal{A}|M^{|\mathcal{A}||\Theta|}.$$

For each of these regions, a single set of constraints is constructed, but many LPs actually need to be solved for each set. Remember that for each set of constraints generated, we need to check each individual constraint to see whether or not it is binding on the region defined by the set of constraints. Therefore, the number of constraints and the number of LPs solved will be exactly the same. This ignores any of the previously described optimizations that can be employed with this algorithm .

The number of constraints (and number of LPs solved) is

$$|\mathcal{A}| + |\mathcal{A}||\Theta|M - |\mathcal{A}||\Theta|.$$

The first term is for the $\alpha_a(t)$ constraints. Although there is one less than that numer of actions (for the $\alpha_a(t)$ that is actually $\alpha^*(t)$), we need to add one for the simplex constraint so that the end result is $|\mathcal{A}|$ constraints. The last two terms are for the $\alpha(t-1)$ constraints. The second term is the total number of ways these constraints can be formed and the last term corrects for the the cases when $\alpha^k(t-1) = \alpha^{\iota(\pi,a,\theta)}(t-1)$. So for a single region, the total number of constraints is the the number of constraints (above) squared.

This number is large and can be significantly trimmed by implementing some of the clever optimizations discussed earlier, but the shear number of regions that could be generated will still dominate this and so, for large problems, we should expect worst case times that are predominantly dictated by $M^{|\mathcal{A}||\Theta|}$ which is actually worse than Monahan's "exhaustive" algorithm which is dominated by only $M^{|\Theta|}$.

4.9.4 Cheng's Relaxed Region Algorithm

This algorithm is the one that is very close to Sondik's One Pass algorithm and so we choose to only discuss the differences in complexity from that algorithm. Since Cheng defines regions which are typically larger than they need be, he is able to stay away from the explosive growth of regions that Sondik's algorithm suffers from. The Relaxed Region algorithm will only require $N = |\mathcal{V}_t^*|$ regions to be defined which is a significant savings. The number of constraints on these regions is also less, though not significantly. For each region the number of constraints is

$$|\mathcal{A}| + |\Theta|M - |\Theta|.$$

The next step in Sondik's algorithm would be to find the binding constraints, but Cheng opts to find all of the corner points of the region. The region defined by these constraints form a convex polytope in $|\mathcal{S}|$ dimensions. Cheng uses an interior point method to find these vertices, and this is where Cheng's solution suffers a bit in complexity. The worst case time for finding all corner points in a convex polytope is exponential in the number of faces. The number of faces in the polytope will be dictated by the smaller of either the number of constraints or the number of variables. Although this could be bad, the huge savings in the number of regions makes this algorithm much more practical[10] to use than either Sondik's or Monahan's.

4.9.5 Cheng's Linear Support Algorithm

Like the Relaxed Region algorithm, this one will only need to specify $N = |\mathcal{V}_t^*|$ different regions. The constraints on the regions will always be the number of vectors in the current approximation, $\hat{\mathcal{V}}_t^*$, along with the simplex constraint. It is difficult to specify how many constraints there will be at each point since it will depend on the nature of the problem and the order in which the regions are processed. Even the number of constraints on the first region will not be known before hand, since the $|\mathcal{S}|$ belief simplex corner points can generate anywhere from 1 to $|\mathcal{S}|$ vectors in the initial approximation. However, at all times we do know an upper bound, since we will never have more than N vectors in our approximation. By definition, if we have N vectors in our approximation, $\hat{\mathcal{V}}$, then we have all of the vectors necessary to exactly specify $V_t^*(\cdot)$ an no more work needs to be done.

Also, like the Relaxed Region algorithm, Linear Support uses an interior point method for finding all of the vertices of the region (i.e., convex polytope) and as a result, inherits the same drawbacks. As stated before, the number of vertices can be exponential in either the number of constraints, at most N, or the number of variables, $|\mathcal{S}|$.

4.9.6 Witness Algorithm

This algorithm's main advantage over Cheng's Linear Support algorithm is that it does not employ an interior point method to enumerate all of the vertices of the regions/volumes defined by a set of constraints. Unfortunately, we are unable to show that it is theoretically better than Cheng's, though empirical results hint at this result.

The analysis of the Witness algorithm requires notation for the sizes of the $\mathcal{Q}_t^a$ sets, since we will discuss the complexity of constructing these sets. We will use N^a to represent the size of the set $\mathcal{Q}_t^a$. An important relationship to understand is that between the size of the $\mathcal{Q}_t^a$ and $\mathcal{V}_t^*$ sets. Unfortunately, this eludes us at this time. We would like to be able to restrict the size of $\mathcal{Q}_t^a$

[10] We use this term loosely.

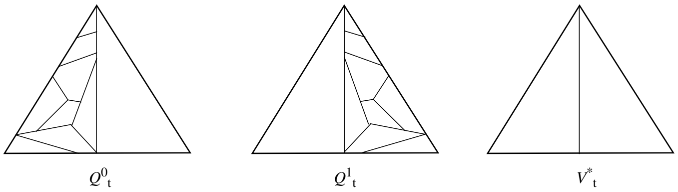

Figure 36: Belief space partitions to illustrate the relationship between the sizes of $\mathcal{Q}_t^a$ and $\mathcal{V}_t^*$.

to be at worst polynomial in the size of $\mathcal{V}_t^*$, since this could be used to show that the Witness algorithm is theorectically better than all the rest. However, we cannot make any such claim. Figure 36 shows a concocted example where the $\mathcal{Q}_t^a$ sizes are significantly larger than $\mathcal{V}_t^*$. The figure shows the partitions imposed by each set on the belief space. Remember that we will combine the $\mathcal{Q}_t^0$ and $\mathcal{Q}_t^1$ and eliminate useless vectors to form $\mathcal{V}_t^*$. In this example only two of the vectors are actually useful, one from each $\mathcal{Q}_t^a$ set.

An analysis of the Witness algorithm is a two step process since the algorithm itself requires two distinct phases. The first phase is the creation of the $\mathcal{Q}_t^a$ sets and the second the elimination of useless vectors from the union of all of these. The second phase is exactly the same as the second step of Monahan's algorithm and we refer to that section for its analysis. Note that the analysis requires knowing the number of vectors in the union of the $\mathcal{Q}_t^a$ sets, which we can say nothing about at this time.

We now describe the analysis for creating one of the $\mathcal{Q}_t^a$ sets. The outer most loop of this algorithm is over the vectors in the approximation which is at worst N^a. For each of these vectors we will need to set up at least $M|\Theta|$ LPs since the algorithm requires trying each vector for every combination of previous vector, $\alpha^k(n-1) \in \mathcal{V}_{t-1}^*$, and observation, θ. An important detail that cannot be missed is that if a new vector is generated from one of these combination, we need to try that combination again. There is an upper bound of N^a on the number of times we could try a particular combination of k and θ, since we can only generate as many vectors as exist in $\mathcal{Q}_t^a$. However, a worst case analysis along these lines would be far off from the actual case.

We can use an amortized cost type analysis to get a better worst case bound on the number of LPs that need to be solved. All vectors, except the first, that are added to the approximation are the result of some combination of an $\hat{\alpha}(t) \in \hat{\mathcal{Q}}_t^a$, k and θ having a non-zero objective function LP. In addition, we know that we are done with a vector $\hat{\alpha}(t)$ when all of the combinations of k and θ have zero objective function LPs. Therefore, we can view the total number of LPs that need to be solved as

$$N^a \left(1 + M|\Theta|\right),$$

since each vector, N^a, requires one LP to discover it and $M|\Theta|$ LPs to determine that we are done with it.

With this upper bound on the number of LPs, we turn to the simpler analysis of the size of the LPs. The number of variables is $|\mathcal{S}|+1$, since there is one for every state in addition to the added δ variable. The number of constraints is at worst

$$N^a + 1.$$

The constraints that restrict the belief region to those π for which $\hat{\alpha}(t)$ is maximum is the number of vectors in the current approximation, which in the worst case is N^a. There is actually one less than this since we do not need to include the constraint that would be generated by the chosen $\hat{\alpha}(t)$, which is also in the approximation. Other than these, there are just two other constraints; the constraint that states that we want an improvement in value and the simplex constraint.

So if we examine the worst case total sum of all the constraints generated in constructing $\mathcal{Q}_t^a$, we would have

$$N^a \left(1 + M|\Theta|\right) \left(N^a + 1\right).$$

We can multiply this whole quantity by $|\mathcal{A}|$ to get the total amount of work to generate all of the sets, but it will require replacing N^a with the largest of all the $\mathcal{Q}_t^a$ set sizes (in order to keep it as a worst case analysis).

An important theoretical note is that the amount of work required to construct $\mathcal{Q}_t^a$ is polynomial in all the quantities of concern: $|calS|$, $|calA|$, $|\Theta|$, $|\mathcal{V}_{t-1}^*|$ and $|\mathcal{Q}_t^a|$. Notice that the size of the $\mathcal{Q}_t^a$ set is itself included. This is because we do not make any claim about how big this set can get with respect to the problem parameters. However, it does say that if the set is small, then we will not do an exorbitant amount of work to construct it.

In addition, the complexity of the elimination of useless vectors from the union of all the $\mathcal{Q}_t^a$ sets is polynomial in those variables of concern. If we could guarantee that the size of the $\mathcal{Q}_t^a$ sets were polynomial in $|calS|$, $|calA|$,

$|\Theta|$ and $|\mathcal{V}^*_{t-1}|$, we would then have a very efficient algorithm. However, such a guarantee eludes us at this time and we must settle for empirical comparisons to the existing algorithms.

4.10 Complications

This section presents some of the implementation complications that we have encountered in our implementations which are not specific to any one algorithm.

As the problem size increases, the value function will typically be comprised of a larger number of vectors. As this number increases the difference between vectors becomes harder to distinguish. The vectors begin to get closer and closer to each other as the solution requires minute differences to differentiate between different regions of the policy. The floating point precision of the existing computers will impose an upper limit on the size of problems that can be solved. After so many decimal places, two slightly different vectors will appear equal to the precision of the machine. Even for modest sized problems this becomes an issue, because there are many instances in the algorithms where these vectors must be compared to each other. Unless extreme care is taken during implementation, this could result in poor algorithms. Round off errors can lead to duplicate slightly different vectors when there should only be one. But if you try to make your implementation distinguish these errors and ignore them, you might be classifying two different vectors as being the same since there difference is not that great.

The heavy dependence on linear programming techniques presents many difficulties. None of these algorithms should be attempted without a stable, robust LP package. Among the more serious problems encountered are those of precision and scaling.

The precision that the LP software uses internally needs to be consistent with those used and assumed by the rest of the implementation. This is closely related to the machine precision problems.

The LPs generated by the algorithms have constraint coefficients that are based upon the differences between two vectors. As the problem size increases, these vectors will get closer and closer in value making the differences extremely small. This results in LPs with a wide dynamic range of coefficients which causes the LPs to be extremely unstable. Good LP packages will scale the constraints to avoid this instability. If the package used does not provide this functionality, erratic and even incorrect answers can result.

5 Conclusions

5.1 Advantages of POMDPs

In this section we try to show why the POMDP model is worth exploring and why deriving optimal solutions is desirable. Obviously, for the latter, optimality is always more desirable (all else being equal) than sub-optimal policies. Even when the computational costs of finding optimal policies is prohibitive, knowing the form and methods for finding the optimal policy can help guide the process of exploring the space of approximate solution techniques.

Another way in which exact solution techniques might be useful is as a sub-component of a larger system. For large world models or environments, finding optimal solutions for every decision process will not be feasible. However, there might be small sub-problems that can be cast in a compact POMDP model, for which we will be able to obtain the optimal policy by these methods.

5.2 Drawbacks/Assumptions of POMDPs

In this section we attempt to list the major problems with POMDP models and exact solutions procedures for these models. One of the most glaring deficiencies is the assumption that the agent knows the complete POMDP model. This requires knowing a vast amount of information (e.g., transition probabilities, observation probabilities, rewards, etc.) and all of this supposes that the agent also knows the number of states, actions and observations. This is not realistic for many problems and it is highly desirable to have algorithms that perform well without having access to such complete information about the environment.

Another limitation is that assumption of finiteness in the model. The POMDP model assumes finiteness of the states, actions and observations, but many problems are better modelled with continuous quantities for these. A robot that has a `turn-left` action, can usually be given any angle not a finite set of angles. The same would be true for a robot's location and its sensory readings.

The solution procedures for POMDPs assume that the model is static. It does not permit the model to change over time. Actually, we can model change over time in this framework, but it requires exploding the state space to deal with all the possible changes that could occur. While theoretically feasible, this state space expansion is impractical and leads to a model of the world which is very unintuitive.

While the notion of a belief state proves to be convenient for some aspects of the model, it requires time to compute this at run time. For certain applications, the resources to maintain and update belief states might not be available.

The most depressing news of solving the POMDP model is that it is impossible to compute the optimal policy for anything but small problems.

5.3 Contributions

The existing literature on POMDPs suffers from a few problems which this work has addressed. The papers that present specific algorithms [17, 16, 6, 4] are difficult to follow unless the reader already has an intimate knowledge of the area . The survey articles [12, 9] are at too high level to give much insight into the details of POMDPs and the solution procedures. To compound this problem, each author uses different notation and terminology than the others. This makes exploring this field very difficult for persons not already familiar with the concepts and results in this area.

This paper has tried to present some of the major work in the POMDP area using consistent notation and terminology. The paper's length is a direct result of our attempt to explain the algorithms and intuitions behind them in language that can be understood by persons not already familiar with this area. We have presented the structure of POMDP solutions in many forms and have tried to relate each of these to the others. We believe our approach is helpful for understanding POMDPs and we have not seen it done this way in any other work to date. This paper has also attempted to sort out and clarify many of the existing bugs in the literature.

There is a related document, [8], that presents the Witness algorithm in a more rigorous and formal setting. It provides many of the derivations and proofs that are required to show that the algorithm is correct. In addition, it also delves into the details about the approximation bounds when exact solutions cannot be found. It would be the preferred document for those who are more familiar with MDPs and POMDPs.

We have also presented a new algorithm, Witness, for computing optimal solutions to finite horizon POMDPs which does not suffer from some of the problems of previous solution techniques. Empirically we have been able to solve problems much larger than any other previously presented work.

The Operations Research community, where most of the POMDP work has come from, does not address the complexity issues when comparing algorithms. In this work we have attempted to classify a number of finite horizon algorithms by their complexity. Though at an extremely crude level, this analysis identifies the limiting elements of the running times of each and allows comparison between them. This rough analysis coincides with the existing empirical performance results, based upon our results and those of [4].

References

[1] K. J. Astrom. Optimal control of markov decision processes with incomplete state estimation. *J. Math. Anal. Appl.*, 10:174–205, 1965.

[2] L. E. Baum. An inequality and associated maximization technique in statistical estimation for probabilistic functions of a markov process. *Inequalities 3*, pages 1–8, 1972.

[3] Richard Bellman. *Dynamic Programming.* Princeton University Press, Princeton, New Jersey, 1957.

[4] Hsien-Te Cheng. *Algorithms for Partially Observable Markov Decision Processes.* PhD thesis, University of British Columbia, British Columbia, Canada, 1988.

[5] A. W. Drake. *Observation of a Markov Process Through a Noisy Channel.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1962.

[6] James N. Eagle. The optimal search for a moving target when the search path is constrained. *Operations Research*, 32(5):1107–1115, 1984.

[7] Ronald A. Howard. *Dynamic Programming and Markov Processes.* The MIT Press, Cambridge, Massachusetts, 1960.

[8] Michael L. Littman. The witness algorithm for solving partially observable markov decision processes. Technical report, Brown University, Providence, Rhode Island, 1994.

[9] William S. Lovejoy. A survey of algorithmic methods for partially observed markov decision processes. *Annals of Operations Research*, 28(1):47–65, 1991.

[10] T. H. Mattheis. An algorithm for determining irrelevant constraints and all verticies in systems of linear inequalities. *Operations Research*, 21:247–260, 1973.

[11] T. H. Mattheis and David S. Rubin. A survey and comparison of methods for finding all vertices of convex polyhedral sets. *Mathematics of Operations Research*, 5(2):167–185, 1980.

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

[13] Sraban Mukherjee and Kiran Seth. A corrected and improved computational scheme for partially observable markov processes. *INFOR*, 29(3):206–212, 1991.

[14] Anton Schwartz. A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the Tenth International Conference on Machine Learning*, Amherst, Massachusetts, 1993. Morgan Kaufmann.

[15] Satinder Pal Singh, Tommi Jaakkola, and Michael I. Jordan. Model-free reinforcement learning for non-markovian decision problems. In *Proceedings of the Machine Learning Conference*, 1994. To appear.

[16] Richard D. Smallwood and Edward J. Sondik. The optimal control of partially observable markov processes over a finite horizon. *Operations Research*, 21:1071–1088, 1973.

[17] Edward J. Sondik. *The Optimal Control of Partially Observable Markov Processes*. PhD thesis, Stanford University, Stanford, California, 1971.

[18] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[19] Wayne L. Winston. *Introduction to Mathematical Programming: Applications and Algorithms*. PWS-KENT, Boston, Massachusetts, 1991.