The Witness Algorithm: Solving Partially Observable Markov Decision Processes

Michael L. Littman

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-94-40
December 1994

The Witness Algorithm: Solving Partially Observable Markov Decision Processes

Michael L. Littman*

December 30th, 1994

1 Introduction

Markov decision processes (MDP's) [Bellman, 1957] are a mathematical formalization of problems in which a decision-maker, or *agent*, must choose how to act to maximize its reward over a series of interactions with its *environment*. Partially observable Markov decision processes (POMDP's) [Drake, 1962, Astrom, 1965, Smallwood and Sondik, 1973] generalize the MDP framework to the case where the agent must make its decisions in partial ignorance of its current situation.

This paper describes the POMDP framework and presents some well-known results from the field. It then presents a novel method called the witness algorithm for solving POMDP problems and analyzes its computational complexity. We argue that the witness algorithm is superior to existing algorithms for solving POMDP's in an important complexity-theoretic sense.

1.1 Application scenario

Figure 1 illustrates an extremely simple robot task. A robot wanders the halls of an office building. Its job is to ensure that the laser printer (P) is stocked with paper. The designers of the robot have guaranteed us that the robot can only be in one of a finite number of states any time a decision is possible. A

*This is joint work with Tony Cassandra and Leslie Kaelbling.

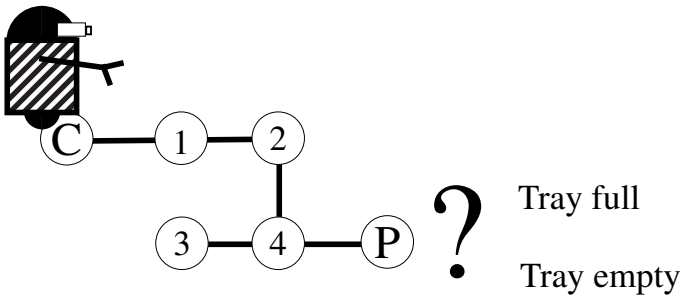

Figure 1: A sample office building task.

state consists of the position of the robot, discretized to the nearest corridor intersection, and the status of the printer. The robot's actions are chosen from a small finite set: N, S, E, and W. Whenever the robot is in the printer room and the tray is empty, the robot's low-level programming kicks in and causes it to refill the tray. Thus, all the robot needs to do is to show up in the printer room each time the paper runs out.

There are two subtleties that make the problem interesting. One is that the robot is penalized one point for each step it is away from its charger (C). It is charged ten points for each step the printer tray remains empty. Ideally, the robot should move directly from the printer room to its charger, returning as soon as the tray is empty (which happens with some random probability on each step).

However, this strategy is not possible because the environment is *partially observable*. That is, from outside the printer room, the robot is unable to determine if the printer tray is empty or full. The optimal policy for the robot is to wait at the charger bay until it expects that more points are being lost due to the tray being empty than it would cost to go and check.

The POMDP problem comes down to taking a map of the environment which includes state transition information, observation probabilities, and the reward structure, and generating a plan for action that maximizes the reward.

1.2 Notation

A POMDP problem can be defined by a finite set of states, $\mathcal{S}$, a finite set of actions, $\mathcal{A}$, and a finite set of observations, $\mathcal{O}$.

Three functions relate these sets. The transition function, $T : \mathcal{S} \times \mathcal{A} \rightarrow \Pi(\mathcal{S})$ defines the effects of the various actions on the state of the environment. ($\Pi(\cdot)$ represents the set of discrete probability distributions over a given finite set.) The notation $T[s, a, s']$ represents the probability that state s' will result from taking action a in state s (that is, $\Pr(s'|s, a)$).

The reward function, $R : \mathcal{S} \times \mathcal{A} \rightarrow \Re$, specifies the agent's payoffs. The payoff is a random function of the state and action with $R[s, a]$ representing the expected immediate reward for taking action a from state s.

The function $O : \mathcal{S} \times \mathcal{A} \rightarrow \Pi(\mathcal{O})$ specifies the observation model. That is, $O[s', a, o]$ is $\Pr(o|s', a)$, the probability of observing o in state s' after having taken action a.

1.3 Objective

The agent's objective is to generate actions so as to maximize its expected sum of reward. However, this is not completely well-defined as it stands. If the agent can guarantee itself a huge reward tomorrow by doing nothing today, what's to keep it from doing nothing forever? This is sometimes called the problem of the *infinitely delayed splurge* [Platzman, 1977] and a standard way to combat it is to use geometric discounting. This means a reward received t steps in the future is only worth γ^t as much as it would be if it were received today. The variable γ is a factor strictly between 0 and 1 that controls how much future rewards are worth[1]. Small values of γ (like 0.01) make future rewards worth very little whereas larger values (like 0.99) make future rewards worth more. Typical values in our experiments are 0.75 and 0.90.

The introduction of the discount factor makes the notion of an optimal policy well-defined. The agent should choose actions to make the expected sum of discounted future rewards,

$$E\{\sum_{i=0}^{\infty} \gamma^i R[s_i, a_i]\},$$

as large as possible. The boundedness of R and the fact that $\gamma < 1$ insure that the value of this infinite sum is finite.

[1]Setting $\gamma = 0$ leads to a simpler problem of maximizing immediate reward. That class of problems is not addressed here because it complicates several of the arguments. All the fundamental results hold in the $\gamma = 0$ case.

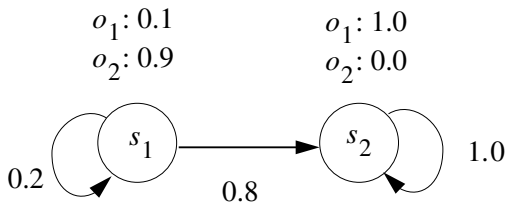

Figure 2: A small POMDP.

1.4 Implementation

This paper describes the witness algorithm for solving POMDP problems. Throughout the paper, pseudocode is given for all critical functions. These fragments, taken together, should be sufficient to allow an experienced programmer to implement the algorithm in a high-level language.

The only difficult procedure that is left unspecified is `solveLP`, which is intended to take a set of variables, linear constraints on those variables, and an objective function, and return bindings for the variables that satisfy the constraints and maximize the objective function (that is, it solves a linear program). Many software packages are available for this.

The argument $\mathcal{M}$, passed to several of the routines, represents the POMDP model we are trying to solve. The model consists of the sets and functions described earlier in this section. So if $\mathcal{M}$ is in the argument list for a function, $\mathcal{S}$, $\mathcal{A}$, T, R, $\mathcal{O}$, and O are all available in the function body.

2 Belief states

In some circumstances, the agent will be able to identify its current state unambiguously. However, other times, the best it can do given the information it has is to compute a probability distribution over states.

As an example, Figure 2 shows a small POMDP with 2 states, 2 observations, and 1 action. If the agent knows it is in s_1 at time t and then takes an action and observes o_2, it can be certain that it is has remained in s_1. However, if it observes o_1, it is possible that it is in either s_1 or s_2.

Table 1 summarizes all 4 possible outcomes. These outcomes are mutually exclusive and as such their probabilities add to one.

The probability that the agent is in s_2 given that it observed o_1 is $0.80/(0.02+$

resulting state	observation	probability of event
s_1	o_1	$0.2 \times 0.1 = 0.02$
s_1	o_2	$0.2 \times 0.9 = 0.18$
s_2	o_1	$0.8 \times 1.0 = 0.80$
s_2	o_2	$0.8 \times 0.0 = 0.00$

Table 1: Probability of all outcomes.

$0.80) \approx 0.976$. Therefore, some of the times when this sequence of events happens, the agent is in s_1. Choosing an action that is dangerous if the agent is in s_1 might not be a good idea in this situation.

2.1 Derivation of the belief state update equation

The analysis from the previous section suggests a possible representation for the agent to use while acting in the world. The agent can keep a probability distribution over $\mathcal{S}$ as a representation of where it is at any given time. Then, this *belief state* can be consulted each time an action is chosen.

This section develops a general method for keeping the belief state updated. The next section argues that the belief state is sufficient information for the agent to choose its actions optimally. Because of these facts, the belief state will become a central part of the representation used in solving POMDP's.

A belief state, $b \in \Pi(\mathcal{S})$, is a representation of the agent's current state given its past history of actions and observations. We need a way to update the belief state as new information arises. Notationally, $b[s]$ represents the probability that the agent's current state is $s \in \mathcal{S}$. By the definition of probability distributions, $b[s] \geq 0$ for all $s \in \mathcal{S}$, and $\sum_s b[s] = 1$.

How might we compute the belief state that results from a particular state, action, observation triplet? Let's return, for a moment, to the example in Figure 2. If we begin in state s_1, we move to state s_2 with probability 0.8 and remain in state s_1 with probability 0.2. Now imagine that we observe o_1. It is much more likely that we'd observe o_1 if we were in state s_2 than if we were in state s_1. We can weight the occupation probabilities by the corresponding observation probabilities. That gives us a weight of 0.02 for s_1 and 0.8 for s_2. We can then normalize these values so that they sum to 1.

Although this seems like a sloppy way to use the various probabilities to

update our state, we can use elementary probability theory to show that it does indeed lead to the proper formula. We want to know the belief state, b', that results from starting in belief state b, taking action a and making observation o. Component s' of the belief state can be written:

$$\begin{aligned}
b'[s'] &= \Pr(s'|o,a,b) \\
&= (\Pr(o|s',a,b)\Pr(s'|a,b)) \,/\, \Pr(o|a,b) \\
&= \left(\Pr(o|s',a,b)\sum_{s\in\mathcal{S}}\Pr(s'|a,b,s)\Pr(s|a,b)\right) / \Pr(o|a,b) \\
&= \left(\Pr(o|s',a)\sum_{s\in\mathcal{S}}\Pr(s'|a,s)\Pr(s|b)\right) / \Pr(o|a,b) \\
&= \left(O[s',a,o]\sum_{s\in\mathcal{S}}T[s,a,s']b[s]\right) / \Pr(o|a,b). \qquad (1)
\end{aligned}$$

The denominator normalizes the resulting belief state to add to 1. It can be computed as follows:

$$\begin{aligned}
\Pr(o|a,b) &= \sum_{s'\in\mathcal{S}}\Pr(o,s'|a,b) \\
&= \sum_{s'\in\mathcal{S}}\Pr(s'|a,b)\Pr(o|s',a,b) \\
&= \sum_{s'\in\mathcal{S}}\sum_{s\in\mathcal{S}}\Pr(s',s|a,b)\Pr(o|s',a) \\
&= \sum_{s'\in\mathcal{S}}\sum_{s\in\mathcal{S}}\Pr(s|a,b)\Pr(s'|s,a,b)\Pr(o|s',a) \\
&= \sum_{s'\in\mathcal{S}}O[s',a,o]\sum_{s\in\mathcal{S}}T[s,a,s']b[s]. \qquad (2)
\end{aligned}$$

Thus, each component of the resulting belief state is derived exactly as described above: we project the current belief state forward one step using the transition matrix, T; we weight the resulting states by the probability of making the associated observation; and then we normalize so the resulting distribution adds to 1.

This shows that through relatively simple manipulation of the observation and transition probabilities, new information can be incorporated into the belief state as the agent moves about.

2.2 Sufficiency of belief states

For the agent to keep its belief state up to date, it must start with a known belief state. That is, it must have some initial probability distribution, b_0. One interpretation of a component of the belief state at time t, $b_t[s]$, is that if the agent were to rerun a million times for t steps starting from b_0, then about $1,000,000\ b_t[s]$ times it would end up in state s.

Since b_t summarizes all the information that could be known about the agent's state, its choice of action at time t can be made solely on the basis of b_t. That is, knowing its belief state is sufficient information for the agent to behave optimally. More formal arguments to this effect have been put forth elsewhere [Astrom, 1965]. The machinery needed to derive this formally is beyond the scope of this paper.

The importance of this result to the current work is that the agent's behavior will be specified as a mapping from its current belief state to an optimal choice of action.

3 Value functions for POMDP problems

The *value* of a belief state, b, is the expected amount of reward that is garnered by the optimal policy if the agent is started off in belief state b:

$$V^*(b) = E\{\sum_{t=0}^{\infty} \gamma^t r_t\}.$$

Here, r_t represents the reward received at time t while following the optimal policy. The function V^* is called the *optimal value function* and it is unique [Bertsekas, 1987].

The optimal *policy*, $\pi^* : \Pi(\mathcal{S}) \rightarrow \mathcal{A}$, maps belief states to actions. It is stationary [Bertsekas, 1987], meaning that the optimal action choice for belief state b is constant over time. This leads to the following convenient representation of the optimal value function (for a formal derivation, see [**?**]):

$$V^*(b) = \max_a \left[\sum_s b[s] R[s,a] + \gamma \sum_o \Pr(o|b,a) V^*(b') \right]. \qquad (3)$$

The variable b' denotes the belief state that results from taking action a and observing o from belief state b. It can be computed from b, a, and o by Equation 1.

Equation 3 deserves further explanation. The expression $\sum_s b[s]R[s,a]$ represents the expected reward the agent receives immediately after taking action a from belief state b. One interpretation of this is that, if the agent believes it is in states that have high payoff, it will get high payoff. It appears the agent is being rewarded for simply *believing* it is in a good state. This confusion stems from the use of the word "belief." Because the beliefs are updated according to Equation 1, the agent's belief state can never be the result of a private fantasy.

The correct interpretation is that since $R[s,a]$ is the immediate reward received by an agent in state s for taking action a and $b[s]$ is the probability that the agent is in state s, $\sum_s b[s]R[s,a]$ is just the probability-weighted sum of immediate rewards.

The expression $\sum_o \Pr(o|b,a)V^*(b')$ represents the probability-weighted average value of the resulting belief state. That is, it averages the reward that will be earned by the optimal policy starting from each b' and weights it by how likely b' is. This is all multiplied by γ since this reward can be gathered starting on the next step.

Finally, the $\max_a$ in Equation 3 means that the agent will choose the action, at each belief state, that will maximize its total reward. Note that the same expression can be used to specify the optimal policy.

3.1 Bounding a step of value iteration

The method of *value iteration* [Bellman, 1957] is a way of approximating the optimal value function, V^*. The next few sections define value iteration and show how it can be used to produce arbitrarily good policies.

The idea behind value iteration is that by iteratively computing improved estimates of the optimal value function, we can get an arbitrarily good estimate. A single step of the process starts with an estimate, V, of the optimal value function, and produces a better estimate, V' where V' is defined as follows:

$$V'(b) := \max_a \left[\sum_s b[s]R[s,a] + \gamma \sum_o \Pr(o|b,a)V(b') \right]. \tag{4}$$

This basically treats the equality in Equation 3 as an assignment operator.

At this point in the discussion, it is not apparent that V' can be defined this way, because this expression would have to be evaluated over every belief state, b. For now, consider these as abstract mathematical functions.

Section 4 describes a representation for V and V' that is suitable for computer manipulation.

To show that applying Equation 4 leads to an improved approximation, we need the following lemma (see [Bertsekas, 1987]).

Lemma 1 *(Value Iteration Step): Starting with an approximate value function, V, that differs by at most $e \geq 0$ from the optimal value function, V^*, at every belief state, the function V', returned by a step of value iteration, differs from V^* by at most γe.*

Proof: The fact that V differs from the optimal value function by no more than e at any b can be written:

$$V^*(b) - e \leq V(b) \leq V^*(b) + e. \tag{5}$$

Let a' be any action chosen in Equation 4 and let a^* be any action chosen in Equation 3. It follows that:

$$\begin{aligned} V'(b) &= \sum_s b[s]R[s,a'] + \gamma \sum_o \Pr(o|b,a')V(b') \\ &\geq \sum_s b[s]R[s,a^*] + \gamma \sum_o \Pr(o|b,a^*)V(b'). \end{aligned}$$

This follows because a' is the action that maximizes the expression so any other action (a^*, say) leads to a value that is no larger. Thus,

$$\begin{aligned} V'(b) &\geq \sum_s b[s]R[s,a^*] + \gamma \sum_o \Pr(o|b,a^*)(V^*(b') - e) \\ &\geq V^*(b) - \gamma e \sum_o \Pr(o|b,a^*) \\ &\geq V^*(b) - \gamma e, \end{aligned} \tag{6}$$

from the bound in Equation 5 and the fact that the probabilities sum to 1.

Again, using the bound in Equation 5 we have:

$$\begin{aligned} V'(b) &= \sum_s b[s]R[s,a'] + \gamma \sum_o \Pr(o|b,a')V(b') \\ &\leq \sum_s b[s]R[s,a'] + \gamma \sum_o \Pr(o|b,a')(V^*(b') + e) \\ &\leq \sum_s b[s]R[s,a^*] + \gamma \sum_o \Pr(o|b,a^*)V^*(b') + \gamma e \sum_o \Pr(o|b,a') \\ &\leq V^*(b) + \gamma e. \end{aligned} \tag{7}$$

The third step uses a^* in place of a' in the given expression. This cannot decrease the value because a^* was chosen to maximize the expression. The rest follows from substituting the definition of V^* and the fact that the probabilities sum to 1.

Combining Equations 6 and 7 gives:

$$V^*(b) - \gamma e \leq V'(b) \leq V^*(b) + \gamma e, \tag{8}$$

for all b. Thus, V' is a better approximation to the optimal value function than V is by a factor of γ, proving the lemma. □

3.2 Value iteration bound

The previous section showed that applying a step of value iteration as defined in Equation 4 leads to an improved value function. This section shows that applying it repeatedly gets us a value function that is as close as we'd like to optimal.

The complete value iteration algorithm looks like this. Start with $V_0(b) = 0$. Now, for i from 1 to t, generate V_i from V_{i-1} using a step of value iteration. The following lemma shows that, by using a large enough value of t, we can find a value function arbitrarily close to V^* (see, e.g. [Heyman and Sobel, 1984]).

Theorem 1 *(Value Iteration): After t steps of value iteration on a POMDP with the magnitude of the maximum per-step reward equal to M and discount factor $0 < \gamma < 1$, starting from the zero value function, the final value function differs from the optimal value function by no more than $M\gamma^t/(1-\gamma)$ at any belief state.*

Proof: Let M be the largest magnitude reward possible from any state: $M = \max_a \max_s |R[s,a]|$. If the agent received this reward (positive or negative) on every step, its total expected reward would be $\sum_{i=0}^{\infty} \gamma^i M = M/(1-\gamma)$. Thus, the zero value function, $V_0 = 0$ cannot differ from the optimal value function by more than $M/(1-\gamma)$ at any belief state.

Let V_t be the value function obtained from t applications of the value iteration step. Repeated application of Lemma 1 tells us V_t differs from V^* by at most $M\gamma^t/(1-\gamma)$. □

To use this result, imagine that we'd like to find a value function that is within ϵ of optimal. If M is the maximum single step reward for the POMDP in question, we can rewrite the bound as:

$$t \geq \frac{\log(M) + \log(\frac{1}{\epsilon}) + \log(\frac{1}{1-\gamma})}{\log(\frac{1}{\gamma})}.$$

Running for this number of steps will be adequate.

The function V_t is the value function obtained after t iterations. However, it can also be interpreted as a t-step value function. That is, $V_t(b)$ is the maximum reward that any policy can accumulate in t steps. A POMDP where the objective is to maximize reward over a finite number of steps is called a *finite-horizon* POMDP. Value iteration links the finite-horizon case to the infinite-horizon one.

3.3 Approximate value iteration bound

The analysis given in Sections 3.1 and 3.2 depends on each step of the value iteration algorithm producing an exact answer. This is extremely difficult to achieve in practice. Often it is easier to implement a step of value iteration with a bit of error but in such a way that the maximum error is never more than some tunable parameter, δ. For example, it is often possible to choose the number of bits of precision for all numeric computations and from this derive an error bound for the entire algorithm.

It is important for the value iteration algorithm described to degrade gracefully as a function of the error in each step. This section shows that it does.

More formally, given an approximate value function, V, that differs from the optimal value function by no more than ϵ anywhere, approximate value iteration produces an *approximation* of V' that differs from V' by no more than δ anywhere. The following lemma tells us the result of iterating this procedure for t iterations using δ-approximations at each step.

Theorem 2 *(Approximate value iteration): After t steps of approximate value iteration on a POMDP with maximum per-step reward of M and discount factor γ, starting from the zero value function, and where each step*

produces approximations that are within δ of the result of true value iteration, the resulting value function differs from the optimal value function by no more than $M\gamma^t/(1-\gamma) + \delta/(1-\gamma)$ at any state.

Proof: Consider a single step of approximate value iteration. If V differs from the optimal value function by no more than e, then the value function V' returned by exact value iteration differs from the optimal value function by no more than γe (Lemma 1). By the triangle inequality, the new approximation differs from the optimal value function by no more than

$$\gamma e + \delta \tag{9}$$

anywhere.

In the previous section we saw that $M/(1-\gamma)$ bounds the distance between the 0 function and the optimal value function. Applying Expression 9 t times to this quantity bounds the t-step error:

$$\begin{aligned} t\text{-step error} &\leq \overbrace{\gamma(\gamma(\gamma \ldots (\gamma}^{t} M/(1-\gamma) + \overbrace{\delta) \ldots + \delta) + \delta) + \delta}^{t} \\ &\leq M\gamma^t/(1-\gamma) + \sum_{i=0}^{t-1} \gamma^i \delta \\ &\leq M\gamma^t/(1-\gamma) + \sum_{i=0}^{\infty} \gamma^i \delta \\ &\leq M\gamma^t/(1-\gamma) + \delta/(1-\gamma). \end{aligned}$$

The step that expands the sum to run from 1 to ∞ is not strictly necessary but simplifies the formula for the bound. $\square$

Note that for a fixed δ, it may not be possible to use approximate value iteration to reduce the error below $\delta/(1-\gamma)$. If you'd like the final value function to be within ϵ of optimal, one way to do it would be to let $\delta = \epsilon(1-\gamma)/2$. Then running for

$$t \geq \frac{\log(M) + \log(\frac{1}{\epsilon}) + \log(\frac{1}{1-\gamma}) + 1}{\log(\frac{1}{\gamma})}$$

iterations will be adequate.

3.4 Stopping criterion

When should value iteration be stopped? Theorems 1 and 2 suggest that we compute the number of iterations to run as a function of the largest magnitude reward in R. This approach is not desirable in practice for several reasons.

For one, the initial value function, $V_0 = 0$, usually differs from the optimal value function, V^*, by much less than the conservative $M/(1-\gamma)$ bound used in the lemmas.

More importantly, we need to consider how the approximate value function, V_t will actually be used to control the agent's behavior. The agent will follow the "greedy" policy that says: always choose the action that looks best according to one-step lookahead with V_t:

$$\pi_t(b) = \arg\max_a \left[\sum_s b[s] R[s,a] + \gamma \sum_o \Pr(o|b,a) V_t(b') \right].$$

What effect does error in the value function have on the reward gathered by an agent following the greedy policy? Could it be that small errors are arbitrarily magnified?

Both these issues are addressed by the following stopping criterion: At each iteration, t, compare V_t and V_{t-1} and stop if the maximum difference is small enough.

Theorem 3 [Williams and Baird, 1993, Heyman and Sobel, 1984] tells us the result of using this criterion.

Theorem 3 *(Bellman residual): If the maximum difference between V_{t-1} and V_t (sometimes called the Bellman residual of V_{t-1}) is less than ϵ, then the reward gathered by the greedy policy on either V_{t-1} or V_t never differs from that of the the optimal policy by more than $2\epsilon\gamma/(1-\gamma)$ at any belief state.*

Note that this result is concerned with *values* here and not instantaneous rewards. That means that the total reward that this type of greedy agent would gather is not too far from what the optimal agent would get.

The proof of this will not be included here. The significance of the lemma is that a value iteration algorithm that stops when the Bellman residual is less than or equal to $\epsilon \geq 0$ will produce a good policy with respect to ϵ.

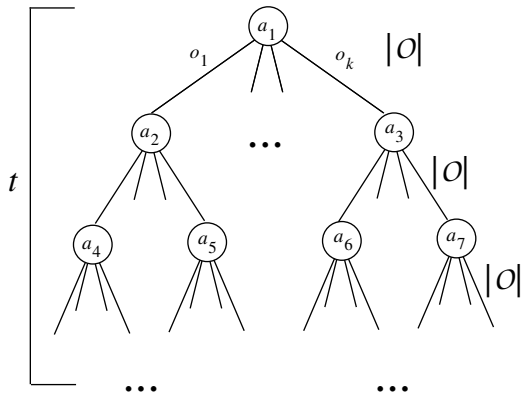

Figure 3: A t-step policy tree.

In combination with Theorem 1, we can argue that this stopping criterion will be satisfied in finite time. In particular, if V_t differs from the optimal value function by no more than $\epsilon/2$ at any b, then (by the triangle inequality)

$$\begin{aligned} |V_t(b) - V^*(b)| &\leq \epsilon/2 \\ |V_{t+1}(b) - V^*(b)| &< \gamma\epsilon/2 \\ |V_{t+1}(b) - V_t(b)| &< \epsilon/2 + \gamma\epsilon/2 \\ |V_{t+1}(b) - V_t(b)| &< \epsilon. \end{aligned}$$

And, by Theorem 1, $|V_t(b) - V^*(b)| \leq \epsilon/2$ in a finite number of iterations.

4 Properties of the finite-horizon value function

Since the discount factor, γ, is less than one and rewards are bounded, we can use value iteration or approximate value iteration to find policies that are within ϵ of optimal by acting according to the value function at time t for large enough t. We therefore need only focus on finding an optimal t-step policy, which is a mapping from all probability distributions over states to optimal actions. This still seems quite imposing, but there is a great deal of structure in the problem we can take advantage of.

The decisions of a t-step policy can be summarized as a *t-step policy tree* like the one in Figure 3. This means that to specify an optimal policy, it is

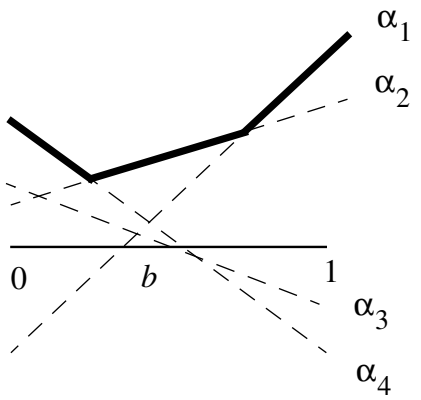

Figure 4: A sample value function for a 2 state POMDP.

sufficient to commit to a t-step policy based on the initial belief state alone. The policy tree chooses an action for each of the t steps as a function of the history of observations.

A fundamental fact about finite-horizon POMDP's is that their value functions are piecewise-linear and convex [Smallwood and Sondik, 1973]. Figure 4 shows an example value function for a simple two state POMDP.

A representation that makes use of these properties was introduced by Sondik [Sondik, 1971].

Theorem 4 *(Piecewise-linear and convex): The optimal t-horizon value function can be written as:*

$$V_t(b) = \max_{\alpha \in \mathcal{C}_t} \sum_s b[s]\alpha[s] \quad ,$$

for some finite size collection, $\mathcal{C}_t$, of $|\mathcal{S}|$-dimensional vectors.

Proof: The vectors in $\mathcal{C}_t$, taken together, specify a value function that is the "upper surface" of a set of linear functions. Smallwood and Sondik [Smallwood and Sondik, 1973] give an algebraic proof that any finite-horizon value function can be represented this way. The following is an alternative derivation that gives an intuitive interpretation to the components of the vectors in $\mathcal{C}_t$. In particular, we can show that the vectors in $\mathcal{C}_t$ correspond to distinct finite-horizon policies.

A t-step policy is completely specified by a choice of action and one $t-1$-step policy for each possible observation. If p is a t-step policy tree, we write

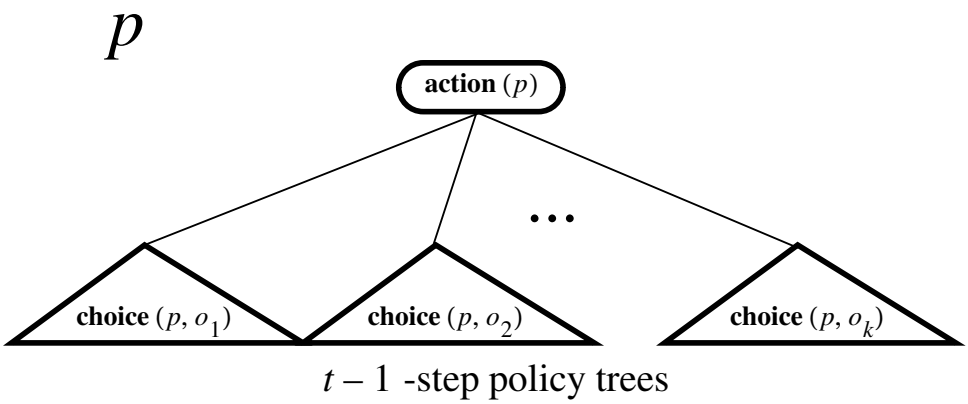

Figure 5: Notation used for describing policy trees.

$\mathbf{action}(p) \in \mathcal{A}$ to signify its initial action choice and $\mathbf{choice}(p, o)$ to be the $t-1$-step policy tree followed if o is observed after the first step. Figure 5 illustrates these definitions.

If we knew our true state at time t, we could compute the expected value due to a particular policy by simply (if somewhat expensively) computing the likelihood of being in each state at each node of the tree and averaging the resulting rewards. For policy tree p, we could write down a vector of values, $\mathbf{value}(p)[s]$, with one component for each state, that gives the expected value for following policy p if our initial state is s.

Of course, we don't know our starting state, but our belief state, b, is exactly the information we need. Because of the linearity of expected values,

$$\sum_s b[s]\mathbf{value}(p)[s] \tag{10}$$

gives the expected reward for tree p starting from belief state b. It combines the probability of being in a given state, s, with the expected reward we'd get if we truly started in s. Thus, for any initial belief state, b, we can compute the value of executing a particular policy tree p.

Notice that the set of all t-step policy trees, $\mathcal{P}_t$, is finite since each tree has a finite number of nodes and there are a finite number of ways of filling in the tree with action decisions. The total number of t-step policy trees is

$$|\mathcal{P}_t| = |\mathcal{A}|^{\frac{(|\mathcal{O}|^t-1)}{|\mathcal{O}|-1}}.$$

Since the set $\mathcal{P}_t$ is finite, we can choose the optimal t-step policy tree for b by evaluating Equation 10 for every tree and choosing the best one:

$$p^* = \arg\max_{p \in \mathcal{P}_t} \sum_{s \in \mathcal{S}} b[s]\mathbf{value}(p)[s].$$

The expression $\sum_{s \in \mathcal{S}} b[s]\mathbf{value}(p^*)[s]$ gives us the value of the optimal t-step policy. Thus, if we let $\mathcal{C}_t$ be the set of vectors corresponding to all possible policy trees,

$$\mathcal{C}_t = \bigcup_{p \in \mathcal{P}_t} \mathbf{value}(p),$$

then we can write the optimal value function for time t as

$$V_t(b) = \max_{\alpha \in \mathcal{C}_t} \sum_{s \in \mathcal{S}} b[s]\alpha[s]. \quad \Box \tag{11}$$

This argument shows that the t-horizon value function is piecewise-linear and convex and also provides us with an (incredibly inefficient) algorithm for solving POMDP's. A more sophisticated algorithm will be presented in later sections that is substantially faster for some problems with "simple" optimal value functions. However, its worst-case bound is no better than the doubly exponential time method described here.

Here is some additional notation to streamline later discussion. First of all, in linear algebra, a dot product is the sum of the componentwise product of two vectors. Thus, for our purposes, $x \cdot y = \sum_s x[s]y[s]$. Secondly, we define the function **best** as

$$\mathbf{best}(b, \mathcal{X}) := \arg\max_{p \in \mathcal{X}} b \cdot \mathbf{value}(p).$$

That is, given a set of policy trees, $\mathcal{X}$, and a vector b, $\mathbf{best}(b, \mathcal{X})$ returns the policy tree in the set whose value vector has the largest dot product with b. This gives us an abbreviated way of writing Equation 11: $V_t(b) = b \cdot \mathbf{value}(\mathbf{best}(b, \mathcal{P}_t))$.

One other piece of notation is:

$$\mathbf{back}(\alpha, a, o)[s] = \sum_{s'} \alpha[s']T[s, a, s']O[s', a, o].$$

Conceptually, **back** returns a vector that consists of the components of α projected backwards one step by action a and observation o. It is used

to combine policy trees together. The expression $\mathbf{back}(\alpha, a, o)[s]$ can be interpreted as the expected reward received by an agent that takes action a starting in state s, observes o and then proceeds using the policy tree corresponding to α. It does not include the reward or discounting from the first step.

Table 2 gives pseudocode for computing **best** and **back**. Note that ties in `best` are not broken arbitrarily. The symbol "$\succ$" is intended to denote a lexicographic ordering over vectors. The rationale and more details are presented in Section 7.5.

5 Value iteration using the policy tree representation

This section describes the basics of using policy trees as the representation of value functions in value iteration.

5.1 Preliminaries

Section 3 discussed the method of value iteration for solving POMDP's without committing to how value functions might be represented. The policy tree representation, introduced in the previous section, can be used in the value iteration computation once a few details have been established.

First, there are two simple improvements that can be made to the method of Section 4 to simplify the computation. First of all, not all the policy trees in $\mathcal{P}_t$ correspond to reasonable strategies. In general, only a small fraction of the **value** vectors for the policy trees in $\mathcal{P}_t$ can ever contribute to the "max" in Equation 11. So instead of considering the set of *all* t-step policy trees, we only ever work with $\mathcal{V}_t \subseteq \mathcal{P}_t$, which is a minimal set of t-step policy trees for representing V_t.

To make this more precise, we call a policy tree *extraneous* if it is not needed to represent the optimal value function. Figure 6 illustrates the three primary ways that a policy tree p_2 might be extraneous. First of all (Figure 6(a)), it might be strictly dominated by some other policy tree or group of policy trees. That is, there is no belief state at which that policy tree is the best. Secondly (Figure 6(b)), there might be only a single point or zero area region at which it is not dominated. This means there is a belief

```
best(b, 𝒳) := {
    bestpol := nil
    bestvec := [−∞, −∞, ..., −∞]
    bestval := −∞
    foreach pol in 𝒳 {
        vec := value(pol, ℳ)
        val := b · vec
        if ((val > bestval) or ((val = bestval) and (vec ≻ bestvec))) {
            bestpol := pol
            bestvec := vec
            bestval := val
        }
    }
    return bestpol
}

back(α, a, o, ℳ) := {
    Let α′ be an uninitialized |𝒮|-vector
    foreach s in 𝒮
        α′[s] := Σ_{s′} α[s′]T[s, a, s′]O[s′, a, o]
    return α′
}

value(p, ℳ) := {
    if (value(p) undefined) {
        a := action(p)
        foreach s in 𝒮
            value(p)[s] := R[s, a] + γ Σ_o back(value(choice(p, o), ℳ),
                                                a, o, ℳ)[s]
    }
    return value(p)
}
```

Table 2: Pseudocode to compute **best**, **back**, and **value**.

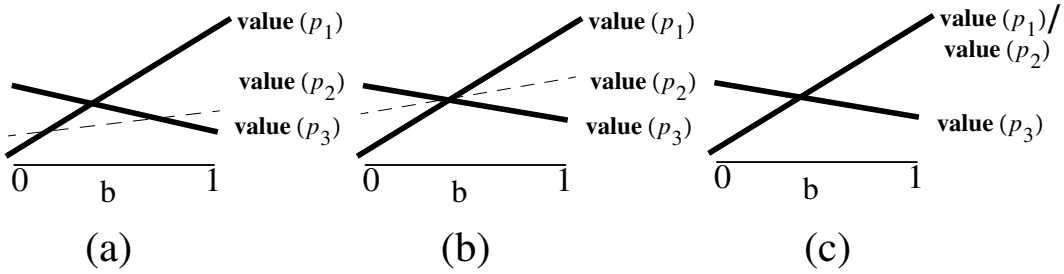

Figure 6: Three ways in which a policy tree p_2 can be extraneous: (a) strict domination, (b) optimal over zero area, (c) tied with existing policy tree.

state at which that policy tree is optimal but some other policy tree covers a strictly large set of states. Lastly (Figure 6(c)), it might be exactly tied with some other policy tree since it is possible for two policy trees to have the same vector representation value. It is possible to argue that every set of policy trees that contains no extraneous policy trees is of minimum size. See Section 6.1 for a few more details.

Another important improvement is the method of computing a policy tree's value vector. As it is described in the previous section, this computation would involve computing expected values over the entire tree, which would take time exponential in its depth. Instead, we can construct the vector for p from the vectors associated with p's subtrees[2]. This is accomplished as follows (using the notation introduced at the end of Section 4).

$$\mathbf{value}(p)[s] = R[s,a] + \gamma \sum_o \mathbf{back}(\mathbf{value}(\mathbf{choice}(p,o)),a,o)[s], \qquad (12)$$

where $a := \mathbf{action}(p)$. This roughly corresponds to the sum of each of the $\mathbf{value}(\mathbf{choice}(p,o))$ vectors projected backwards one step and discounted with the immediate reward added in. The derivation of this is given in Section 7.3. Pseudocode appears in Table 2.

Since the t-step policy trees are constructed from $t-1$-step policy trees and some of the $t-1$-step policy trees have been thrown away, a natural question is whether any of the policy trees that have been thrown away might be needed later on. The answer to this is no. The only way it would

[2] As long as these quantities are cached, i.e. dynamic programming is used, this computation is extremely efficient.

make sense to use a particular $t-1$-step policy tree in the construction of a t-step policy tree is if there would be *some* belief state for which the $t-1$-step policy tree is optimal (i.e., if the policy tree is not extraneous). Otherwise, it would be better to transfer control to the better $t-1$-step policy tree. So if a policy tree is deemed extraneous, it will never be needed again.

5.2 Computing an optimal policy tree for a belief state

Equation 12 can be used to generate a superset of the non-extraneous t-step policy trees by considering all possible combinations of the non-extraneous $t-1$-step policy trees. The extraneous t-step policy trees that are generated can then be identified and removed using the techniques discussed in Section 7.4. This approach, first introduced by Monahan [Monahan, 1982], can be very efficient when the number of observations is small.

To find the minimal representation of the t-step value function, a more important task is to construct, from the set of non-extraneous $t-1$-step policies, $\mathcal{V}_{t-1}$, the optimal t-step policy tree for a given belief state, b. We'll call this t-step policy tree p^* and it is optimal for at least one belief state. However, because many policy trees might be equally good from a given b, we need to take special care to ensure that the p^* generated is not extraneous. Section 7.5 discusses this issue in more detail.

One method for finding p^* is first to consider the policy tree p_a defined to be the policy that takes action a from belief state b and then proceeds optimally thereafter. By definition, $\mathbf{action}(p_a) = a$. Then, after observing o, the agent would be in belief state $b'_{a,o}$, say, and so should follow the $t-1$ step policy tree $\mathbf{choice}(p_a, o) = \mathbf{best}(b'_{a,o}, \mathcal{V}_{t-1})$ thereafter. By substituting and rearranging, we find that:

$$\begin{aligned}
\mathbf{choice}(p_a, o) &= \mathbf{best}(b'_{a,o}, \mathcal{V}_{t-1}) \\
&= \arg\max_{p' \in \mathcal{V}_{t-1}} b'_{a,o} \cdot \mathbf{value}(p') \\
&= \arg\max_{p' \in \mathcal{V}_{t-1}} \sum_{s'} b'_{a,o}[s'] \mathbf{value}(p')[s'] \\
&= \arg\max_{p' \in \mathcal{V}_{t-1}} \sum_{s'} \frac{(O[s', a, o] \sum_s T[s, a, s'] b[s]) \, \mathbf{value}(p')[s']}{\Pr(o|a, b)}
\end{aligned}$$

$$
\begin{aligned}
&= \arg\max_{p' \in \mathcal{V}_{t-1}} \sum_s b[s] \sum_{s'} O[s', a, o] T[s, a, s'] \mathbf{value}(p')[s'] \\
&= \arg\max_{p' \in \mathcal{V}_{t-1}} b \cdot \mathbf{back}(\mathbf{value}(p'), a, o) \\
&= \mathbf{best}(b, \{\mathbf{back}(\mathbf{value}(p'), a, o) : p' \in \mathcal{V}_{t-1}\}). \qquad (13)
\end{aligned}
$$

This says that, after taking action a and making observation o from belief state b, an agent should choose the $t-1$-step policy tree, p', whose value projected backwards one step has the largest dot product with b. The backward projection is needed to allow us to compare b, a t-step belief state, with p', a $t-1$-step policy tree.

Now we've defined p_a for each a, where p_a represents the optimal policy tree to choose from b given that a is the first action. Then p^*, the optimal policy tree to take starting from b, is simply the best of the p_a policy trees,

$$p^* = \mathbf{best}(b, \{p_a : a \in \mathcal{A}\}).$$

The pseudocode in Table 3 returns p_a, an optimal t-step policy tree given an initial action, a, and belief state, b. To avoid constructing extraneous policy trees, ties are broken in a slightly unusual way. See Section 7.5 for an explanation.

6 The witness algorithm

This section develops an exact algorithm for performing value iteration in POMDP problems. The name "witness" comes from the technique we use for determining when we have an exact representation of a value function.

6.1 Basic approach

A single step of value iteration involves taking a representation of V_{t-1} and a POMDP model, $\mathcal{M}$, and returning a representation for V_t. As described in the previous section, we can represent both V_{t-1} and V_t using collections of policy trees, which we'll write as $\mathcal{V}_{t-1}$ and $\mathcal{V}_t$, respectively. Although these representations are not necessarily unique, as long as they contain no extraneous policy trees they are guaranteed to be of minimum size. This can be shown using a fairly straightforward application of matroid theory but is beyond the scope of this paper.

```
besttree(b, a, V_{t-1}, M) := {
    Let p* be a new policy tree
    action(p*) := a
    foreach o in O {
        bestpol' := {}
        bestvec' := [-∞, -∞, ..., -∞]
        bestval := -∞
        foreach pol' in V_{t-1} {
            vec' := back(value(pol', M), a, o, M)
            val := b · vec'
            if ((val > bestval) or ((val = bestval) and (vec' ≻ bestvec'))) {
                bestpol' := pol'
                bestvec' := vec'
                bestval := val
            }
        }
        choice(p*, o) := bestpol'
    }
    return p*
}
```

Table 3: Algorithm to return the best policy tree starting with a given action for a belief state.

There are two classical ways of generating $\mathcal{V}_t$. The first involves generating a superset of the policy trees, $\mathcal{V}_t^+$, and then deleting those that are extraneous [Monahan, 1982, Eagle, 1984]. The process for deleting extraneous trees is described in Section 7.4. Regardless of the true size of $\mathcal{V}_t$, algorithms based on this approach will generate an exponential (in $|\mathcal{V}_{t-1}|$) number of policy trees.

Another class of algorithms work with the set $\hat{\mathcal{V}}_t \subseteq \mathcal{V}_t$ and search for policy trees in $\mathcal{V}_t - \hat{\mathcal{V}}_t$. That is, they build up to $\mathcal{V}_t$ one policy tree at a time. These approaches are sometimes called *relaxed region algorithms* [Cheng, 1988] because the vectors corresponding to the policy trees in $\hat{\mathcal{V}}_t$ partition belief space into regions that constitute a relaxation (a form of approximation) of the regions corresponding to the true V_t function. The algorithms work by extending the set $\hat{\mathcal{V}}_t$ with non-extraneous policy trees until no more exist at which point $\mathcal{V}_t = \hat{\mathcal{V}}_t$ and the algorithms can terminate.

The earlier algorithms that follow this approach [Cheng, 1988, Sondik, 1971] have worst-case running times that are exponential in the size of $\mathcal{V}_{t-1}$. There is some reason to believe that this is a necessary feature of this problem (see Section 7.1).

In this section we take a slightly different approach. Instead of building up a representation of V_t directly, we first find a collection of policy trees that represents Q_t^a, for each $a \in \mathcal{A}$. These Q-functions are defined analogously to the Q-functions of Watkins [Watkins, 1989]—the expression $Q_t^a(b)$ represents the expected reward for taking action a from belief state b and then acting optimally for the the remaining $t-1$ steps.

More formally, we can write

$$Q_t^a(b) = \sum_s b[s]R[s,a] + \gamma \sum_o \Pr(o|a,b)V_{t-1}(b'), \tag{14}$$

where b' is the belief state resulting from taking action a and observing o from belief state b. Since V is defined to be the value of the best action, we have $V_t(b) = \max_a Q_t^a(b)$. This follows directly from the value iteration equation, Equation 4.

Using arguments similar to those in Theorem 4, we can show that these Q-functions are piecewise-linear and convex and can be represented by collections of policy trees. We'll use $\mathcal{Q}_t^a$ to represent the collection of policy trees that specify Q_t^a. Once again, any representation with no extraneous policy trees is guaranteed to be of minimum size.

Note that the policy trees needed to represent the function V_t are a subset of the policy trees needed to represent all of the Q_t^a functions: $\mathcal{V}_t \subseteq \bigcup_a \mathcal{Q}_t^a$. This is because maximizing over actions and then policy trees is the same as maximizing over the pooled policy trees. Section 7.4 describes a method for removing the $\mathcal{Q}_t^a$ policy trees that are not needed for a minimal representation of V_t.

At any moment in time, the witness algorithm maintains a subset of the complete set of non-extraneous policy trees, $\hat{\mathcal{Q}}_t^a \subseteq \mathcal{Q}_t^a$. In polynomial time *per policy tree* of $\mathcal{Q}_t^a$, the algorithm generates a representation for Q_t^a. For problems in which Q_t^a has a simple representation, the algorithm will be very efficient. However, there are problems for which Q_t^a requires a huge number of policy trees to be represented exactly. Even worse, there are problems in which V_t can be represented succinctly but Q_t^a can not. Like previous algorithms, the witness algorithm can run in time exponential in the size of $\mathcal{V}_t$.

In what sense is the witness algorithm superior to previous algorithms for solving POMDP problems, then? For one thing, preliminary experiments indicate that the witness algorithm is faster in practice (see [Cassandra, 1994] for some very informal results). The primary complexity-theoretic difference is that the witness algorithm runs in polynomial time in the number of policy trees in $\mathcal{Q}_t^a$. There are examples that cause the other algorithms, although they never construct the $\mathcal{Q}_t^a$'s directly, to run in time exponential in the number of policy trees in $\mathcal{Q}_t^a$.

The code in Table 4 outlines our approach to solving POMDP problems.

The basic structure is that of value iteration (Section 3.1) with the stopping criteria defined in Section 3.4 (methods for computing the Bellman residual are given in Section 7.2). At iteration t, the algorithm has a representation of the optimal t-step value function. Within the value iteration loop, separate Q-functions are found for each action, represented by sets of policy trees. The union of these sets forms a representation of the optimal value function. Since there may be extraneous policy trees in the combined set, a separate routine (Section 7.4) is called to simplify the representation of V_t.

```
solvepomdp(ε, M) := {
    V_0 := {}
    t := 1
    do {
        foreach a in A
            Q_t^a := witness(V_{t-1}, a, M)
        V_t := purge(⋃_a Q_t^a)
        t := t + 1
    } until (difference(V_{t-1}, V_t) ≤ ε)
    return V_t
}
```

Table 4: Algorithm to solve a POMDP.

6.2 Witness loop

The basic structure of the witness algorithm is as follows. We'd like to find a minimal set of policy trees for representing Q_t^a for each a. We consider the Q-functions one at a time. The set $\hat{\mathcal{Q}}_t^a$ is a set of non-extraneous policy trees and it is initialized with a single policy tree that is the best for some arbitrary belief state. At each iteration we ask, is there some belief state, b, for which the true value, $Q_t^a(b)$, computed by one-step lookahead, is different from $\hat{Q}_t^a(b)$ computed using the set $\hat{\mathcal{Q}}_t^a$? We call such a point a *witness* point because it can, in a sense, testify to the fact that the set $\hat{\mathcal{Q}}_t^a$ is not yet a perfect representation of $Q_t^a(b)$.

Once a witness is identified, we can use `besttree` (Section 5.2) to find a non-extraneous policy tree that is missing from $\hat{\mathcal{Q}}_t^a$. We include the new policy tree to improve the approximation. This process continues until we can prove that no more witness points exist and therefore that the current approximation is perfect.

Table 5 gives pseudocode for this outer loop. The function `findb` identifies witness points and will be defined later.

$\texttt{witness}(\mathcal{V}_{t-1}, a, \mathcal{M}) := \{$
$\quad \hat{\mathcal{Q}}_t^a := \{\texttt{besttree}([1,0,0,\ldots,0], a, \mathcal{V}_{t-1}, \mathcal{M})\}$
$\quad b := \texttt{findb}(a, \mathcal{V}_{t-1}, \hat{\mathcal{Q}}_t^a, \mathcal{M})$
$\quad$ **while** $(b \neq \texttt{nil})$ $\{$
$\quad\quad \hat{\mathcal{Q}}_t^a := \hat{\mathcal{Q}}_t^a \cup \{\texttt{besttree}(b, a, \mathcal{V}_{t-1}, \mathcal{M})\}$
$\quad\quad b := \texttt{findb}(a, \mathcal{V}_{t-1}, \hat{\mathcal{Q}}_t^a, \mathcal{M})$
$\quad\}$
$\quad$ **return** $\hat{\mathcal{Q}}_t^a$
$\}$

Table 5: Algorithm for finding a representation for a Q-function.

6.3 Identifying a witness

This section describes how we go about finding a witness point. First, we need to define how the policy trees in $\hat{\mathcal{Q}}_t^a$ divide belief space into regions.

Each policy tree, $p \in \hat{\mathcal{Q}}_t^a$ accounts for a region of belief space in the following sense. Define:

$$\textbf{region}(p, \hat{\mathcal{Q}}_t^a) := \{b \in \Pi(\mathcal{S}) : \textbf{value}(p) \cdot b \geq \textbf{value}(\tilde{p}) \cdot b, \text{for all } \tilde{p} \in \hat{\mathcal{Q}}_t^a\}. \tag{15}$$

A policy tree's region is just the set of belief states over which it is optimal (with respect to the current approximation). It should be clear that every point in belief space is in some region and that, along the borders, some belief states are in multiple regions.

For simplicity of notation, we define $\hat{Q}_t^a(b) = \textbf{best}(b, \hat{\mathcal{Q}}_t^a) \cdot b$, the value at b of the current approximation. Further, if p is a t-step policy tree, o an observation, and p' a $t-1$-step policy tree, then we define p_{new} as a t-step policy tree that agrees with p in its action and all its choices except for observation o for which $\textbf{choice}(p_{\text{new}}, o) = p'$. Figure 7 illustrates the relationship between p and p_{new}.

Theorem 5 *(Witness theorem): The true Q-function, Q_t^a, differs from the approximate Q-function, $\hat{Q}_t^a$, if and only if there is some $p \in \hat{\mathcal{Q}}_t^a$, $o \in \mathcal{O}$, and*

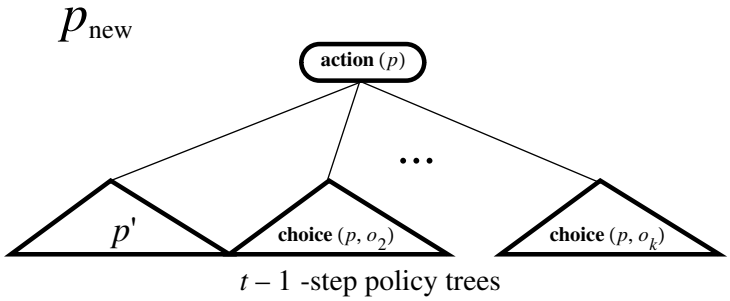

Figure 7: Constructing a new policy.

$p' \in \mathcal{V}_{t-1}$ *for which there is some* $b \in \mathbf{region}(p, \hat{\mathcal{Q}}_t^a)$ *such that*

$$\mathbf{back}(\mathbf{value}(p'), a, o) \cdot b > \mathbf{back}(\mathbf{value}(\mathbf{choice}(p, o)), a, o) \cdot b \qquad (16)$$

In words, if there is a belief state, b, for which we can change the choice made for any observation to a better policy tree, then b is a witness. Conversely, if none of the individual choices can be improved, there are no witness points.

Proof: Define the policy tree p_{new} as above: $\mathbf{action}(p_{\text{new}}) = \mathbf{action}(p) = a$, $\mathbf{choice}(p_{\text{new}}, o) = p'$, and $\mathbf{choice}(p_{\text{new}}, \tilde{o}) = \mathbf{choice}(p, \tilde{o})$, for $\tilde{o} \neq o$. First we show that if Condition 16holds, we've found a witness.

$$\begin{aligned}
\hat{Q}_t^a(b) &= b \cdot \mathbf{value}(p) \\
&= \sum_s b[s] \left(R[s,a] + \gamma \sum_{\tilde{o}} \mathbf{back}(\mathbf{value}(\mathbf{choice}(p, \tilde{o})), a, \tilde{o})[s] \right) \\
&= \sum_s b[s] R[s,a] + \gamma \sum_{\tilde{o} \neq o} \mathbf{back}(\mathbf{value}(\mathbf{choice}(p, \tilde{o})), a, \tilde{o}) \cdot b \\
&\quad + \gamma \mathbf{back}(\mathbf{value}(\mathbf{choice}(p, o)), a, o) \cdot b \\
&< \sum_s b[s] R[s,a] + \gamma \sum_{\tilde{o} \neq o} \mathbf{back}(\mathbf{value}(\mathbf{choice}(p, \tilde{o})), a, \tilde{o}) \cdot b \\
&\quad + \gamma \mathbf{back}(\mathbf{value}(p'), a, o) \cdot b \\
&< b \cdot \mathbf{value}(p_{\text{new}}) \\
&< Q_t^a(b).
\end{aligned}$$

Since we can replace $\mathbf{choice}(p, o)$ with p' and improve the value, Q_t^a and $\hat{Q}_t^a$ differ.

To prove the converse, assume $Q_t^a \neq \hat{Q}_t^a$ and for every p, o, p' and $b \in \mathbf{region}(p, \hat{\mathcal{Q}}_t^a)$ quadruple, Condition 16 is not satisfied. We show that this leads to a contradiction.

If $Q_t^a \neq \hat{Q}_t^a$, it must be the case that $Q_t^a(b) > \hat{Q}_t^a(b)$ for some b. This is because $Q_t^a(b)$ is computed as a maximization over a set of policy trees that is a superset of $\hat{\mathcal{Q}}_t^a$. Let p^* be a policy tree that achieves $Q_t^a(b)$, i.e., $Q_t^a(b) = b \cdot \mathbf{value}(p^*)$. Let p be a policy tree in $\hat{\mathcal{Q}}_t^a$ which is best at b, i.e., $\hat{Q}_t^a(b) = b \cdot \mathbf{value}(p)$. By construction we have $b \cdot (\mathbf{value}(p^*) - \mathbf{value}(p)) > 0$. But,

$$
\begin{aligned}
& b \cdot (\mathbf{value}(p^*) - \mathbf{value}(p)) \\
&= \sum_s b[s] R[s,a] + \gamma \sum_{\tilde{o}} \mathbf{back}(\mathbf{value}(\mathbf{choice}(p^*, \tilde{o})), a, \tilde{o}) \cdot b \\
&\quad - \sum_s b[s] R[s,a] - \gamma \sum_{\tilde{o}} \mathbf{back}(\mathbf{value}(\mathbf{choice}(p, \tilde{o})), a, \tilde{o}) \cdot b \\
&= \gamma \sum_{\tilde{o}} b \cdot (\mathbf{back}(\mathbf{value}(\mathbf{choice}(p^*, \tilde{o})), a, \tilde{o}) \\
&\quad - \mathbf{back}(\mathbf{value}(\mathbf{choice}(p, \tilde{o})), a, \tilde{o})) \\
&\leq 0
\end{aligned}
$$

The last step is justified because we assumed that Condition 16 does not hold for any $p, o, p', b \in \mathbf{region}(p, \hat{\mathcal{Q}}_t^a)$ quadruple (in particular, $p' := \mathbf{choice}(p, \tilde{o})$, $o := \tilde{o}$, $p := p^*$). But this contradicts the fact that $b \cdot (\mathbf{value}(p^*) - \mathbf{value}(p)) > 0$. The conclusion is that, if our approximation is not perfect, there must be some single choice that can be improved. □

6.4 Checking the witness condition

Theorem 5 requires us to search for a $p \in \hat{\mathcal{Q}}_t^a$, an $o \in \mathcal{O}$, a $p' \in \mathcal{V}_{t-1}$ and a $b \in \mathbf{region}(p, \hat{\mathcal{Q}}_t^a)$ such that

$$\mathbf{back}(\mathbf{value}(p'), a, o) \cdot b > \mathbf{back}(\mathbf{value}(\mathbf{choice}(p, o)), a, o) \cdot b$$

or to guarantee that no such quadruple exists. Since $\hat{\mathcal{Q}}_t^a$, $\mathcal{O}$, and $\mathcal{V}_{t-1}$ are finite and hopefully small, checking all combinations is not too time consuming. However, for each combination, we need to check all the belief states in $\mathbf{region}(p, \hat{\mathcal{Q}}_t^a)$. This we can do using linear programming.

For each combination of p, o, p' we compute the vector

$$\beta = \mathbf{back}(\mathbf{value}(p'), a, o) - \mathbf{back}(\mathbf{value}(\mathbf{choice}(p, o)), a, o).$$

For any belief state, b, $\beta \cdot b$ gives the (undiscounted) advantage of following policy tree p_{new} instead of p starting from b. We'd like to find a b that maximizes this advantage under the restriction that $b \in \mathbf{region}(p, \hat{\mathcal{Q}}_t^a)$ (i.e., that we would have followed p in the first place).

The linear program in Table 6 solves exactly this problem. It has a set of region constraints that restrict b to the region over which p is optimal and a set of simplex constraints that force b to be a well-formed belief state. It then seeks to maximize the advantage of p_{new} over p. Since the constraints and the advantage function are all linear, this can be accomplished by linear programming. The total size of the linear program is one variable for each component of the belief state and one constraint for each policy tree in $\hat{\mathcal{Q}}_t^a$ plus one constraint to ensure that the belief state sums to unity.

If the linear program finds that the biggest advantage is not positive, i.e. $\max_b \beta \cdot b \leq 0$, then p_{new} is not an improvement over p. Otherwise, it is and b is a witness point.

Table 7 gives the complete algorithm. It introduces some additional notation for efficiency reasons. The external variable "*checkme*" holds on to the triple p, o, p' that needs to be checked next in `findb`. The function call `nextcheckme`(*checkme*) is intended to return the next triple to check or `nil` if all have been checked. The implementation of these functions is heavily language dependent and so is not addressed in this paper.

Note that in this function, once a region has been searched and no witness point is found, it is never checked again. This is safe because the only thing that changes between successive calls is that new policy trees are added to $\hat{\mathcal{Q}}_t^a$. Additional policy trees can only shrink the existing regions and hence will not introduce any witness points in regions that have already been searched.

The comparison between the objective value and zero in `findb` may seem a little suspect. For the theoretical analysis, we assume that all arithmetic uses arbitrary precision rational numbers and hence a direct comparison with zero is possible. In practice, however, it may be necessary to choose a precision factor, δ, and to compare *objective* to δ instead of to zero. Using $\delta > 0$ is equivalent to running approximate value iteration. See Section 3.3 for an explanation of how this affects the final results.

Inputs:

$\hat{\mathcal{Q}}_t^a, \beta, p$

Variables:

$b[s]$ for each $s \in \mathcal{S}$

Maximize: $b \cdot \beta$

Region constraints:

For each p_2 in $\hat{\mathcal{Q}}_t^a$: $b \cdot \mathbf{value}(p) \geq b \cdot \mathbf{value}(p_2)$

Simplex constraints:

For each $s \in \mathcal{S}$: $b[s] \geq 0$

$\sum_{s \in \mathcal{S}} b[s] = 1$

Table 6: The linear program used to find witness points.

```
findb(a, V_{t-1}, Q̂_t^a, M) := {
    loop:
        (p, o, p') := checkme
        β := back(value(p', M), a, o, M) − back(value(choice(p, o), M), a, o, M)
        LP := set up the witness LP (Table 6) with Q̂_t^a, β, p
        (objective, b) := solveLP(LP)
        if (objective > 0) return b
        checkme := nextcheckme(checkme)
        if (checkme = nil) return nil
        goto loop
}
```

Table 7: Pseudocode for finding a witness point.

6.5 Analysis of running time

This section analyzes the running time of the witness algorithm.

Theorem 6 *(Running time of witness): The running time of a single pass of value iteration using the witness algorithm is bounded by a polynomial in the size of the state space ($|\mathcal{S}|$), the size of the action space ($|\mathcal{A}|$), the number of policy trees in the representation of the previous iteration's value function ($|\mathcal{V}_{t-1}|$), the number of observations ($|\mathcal{O}|$), and the number of policy trees in the representation of the current iteration's Q-functions ($\sum_a |\mathcal{Q}_t^a|$).*

Note that we must assume that the number of bits of precision used in specifying the model, $\mathcal{M}$, is polynomial in these quantities since the running time of linear programming is expressed as a function of the input precision [Schrijver, 1986].

Proof: We would like to bound the running time of a single call to `witness` (Table 5). The work can be divided up as follows.

Each time the witness linear program (Table 6) is solved, either a p, o, p' triple can be discarded, or a witness is found and is turned into a policy tree and added to $\hat{\mathcal{Q}}_t^a$. Thus the total number of linear programs solved in finding the Q-function for action a is exactly

$$|\mathcal{Q}_t^a||\mathcal{O}||\mathcal{V}_{t-1}| + |\mathcal{Q}_t^a| - 1.$$

This is the sum of the number of p, o, p' triples encountered during the computation plus the number of witnesses found (minus one because no linear program is needed to identify the first witness.)

Each of these linear programs has $|\mathcal{S}|$ variables and $|\hat{\mathcal{Q}}_t^a| + 1 \leq |\mathcal{Q}_t^a| + 1$ constraints. The total work due to solving linear programs consists of solving a polynomial number of polynomial-sized linear programs, each of which can be solved in polynomial time [Schrijver, 1986].

The other work performed is primarily in calls to `back` and `besttree`, each of which is trivially implemented to run in polynomial time. Each routine is called only a polynomial number of times: `back` is called twice per linear program and `besttree` is solved once for each witness and therefore each policy tree in $|\mathcal{Q}_t^a|$.

The `witness` function is called $|\mathcal{A}|$ times per iteration and a total of $\sum_a |\mathcal{Q}_t^a|$ policy trees are created and sent to `purge`. Section 7.4 explains how `purge` is implemented in polynomial time.

Thus, the total runtime of a single iteration of value iteration takes polynomial time in the quantities listed. □

7 Odds and ends

This section contains several results needed elsewhere in the paper whose inclusion would have interrupted the flow.

7.1 Comparison to the earlier witness algorithm

We sketched an earlier version of the witness algorithm in a recent paper [Cassandra *et al.*, 1994]. The gross structure of the algorithm presented here is identical to that of the earlier algorithm: both use value iteration to find an approximate representation of the value function using finite-horizon policy trees. The primary difference is that where the current algorithm finds a representation for Q_t^a and creates from that a representation for V_t, the earlier algorithm attempted to solve for V_t directly. Although the difference may seem subtle, it has profound consequences.

The main distinction between the two approaches comes down to the following fact. Consider the *Q-function witness problem* defined as: Given a set of t-step policy trees, $\hat{\mathcal{Q}}_t^a$, that constitutes a partial representation of Q_t^a, and a set of $t-1$-step policy trees, $\mathcal{V}_{t-1}$, that exactly represents the $t-1$-step value function, is there a point, b, at which the partial representation of Q_t^a is not exact? As shown in this paper, this question can be answered in polynomial time.

Surprisingly, the same question applied to V-functions is much more difficult: the corresponding *V-function witness problem* is NP-complete. This means that there is a polynomial time algorithm to solve it if and only if P = NP. This can be shown using a somewhat messy reduction to MAX-2-SAT or quadratic programming [Garey and Johnson, 1979]. Our earlier algorithm claimed to solve the V-function witness problem in polynomial time. In fact, the algorithm runs in polynomial time but will not always give correct answers. It is possible to construct cases in which it fails to find witness points in certain regions of belief space with the result that the algorithm may report "there is no difference between the true V_t and our approximation" when in fact witnesses to this difference exist.

The difficulty of the V-function witness problem stems from the breakdown of the Witness Theorem (Theorem 5) for V-functions. The Witness Theorem tells us that to determine if there is an improvement that can be made to the Q-function, we need only examine a polynomial number of candidate policy trees (those derived from all $|\mathcal{Q}_t^a||\mathcal{O}||\mathcal{V}_{t-1}|$ ways of combining p, o, and p'). On the other hand, no simple test is known for finding improvements in the V-function. It appears that it is necessary to check an exponential number of combinations (all $|A||\mathcal{O}|^{|\mathcal{V}_{t-1}|}$ possible t-step policy trees).

Thus, the extra constraint of holding the first action in the policy constant yields great computational leverage. The downside is that the Q-functions might be a great deal more complicated than the monolithic V-function and therefore might require a great deal more work to derive. But since we allow ourselves time polynomial in the size of the Q-function, this is not counted against us. In a sense, the apparent efficiency of the revised algorithm is a sleight of hand: we are measuring the performance of the two algorithms differently. Nonetheless, we feel bounding the computation by a polynomial in the size of the Q-functions is better than bounding it as an exponential in the size of the V-function since in the case when both functions are small, our running time will be small also.

The NP-completeness result only applies to specific iterative approaches to improving the value function representation. It may be possible, using more of the structure of the problem at hand, to find a representation quickly for the V-function without first generating a representation for each Q_t^a. Such an algorithm has eluded us thus far.

Two additional points are worth making here. First, although the algorithm described in our earlier paper [Cassandra *et al.*, 1994] was not formally correct, the experimental results reported there were valid. We have rerun the same examples with the revised algorithm and found no important differences.

Secondly, although the present algorithm potentially does a great deal more work than the original one, it does have the important advantage that no error tolerance factor need be specified in advance. The witness algorithm is an exact method.

7.2 Computing the Bellman residual

In deciding when the current approximation to the infinite-horizon value function is good enough, the `solvepomdp` algorithm calls `difference` to compute the Bellman residual. Recall that the Bellman residual is the greatest difference between the value functions of two successive iterations.

We have looked at two methods for computing the Bellman residual. One is exact and somewhat expensive. The other is weak but very cheap to compute. We have only implemented the weak method for use in our experiments[3]. There are actually many ways to compute the Bellman residual but since we were satisfied with the weak method in our experiments, we did not pursue this issue very deeply.

7.2.1 An exact method

Briefly, the exact method considers all pairs of policy trees $p_t \in \mathcal{V}_t$ and $p_{t-1} \in \mathcal{V}_{t-1}$. It then uses linear programming to find a $b \in \mathbf{region}(p_t, \mathcal{V}_t) \cap \mathbf{region}(p_{t-1}, \mathcal{V}_{t-1})$ (if it is non-empty) such that $|b \cdot (\mathbf{value}(p_t) - \mathbf{value}(p_{t-1}))|$ is maximized. The maximum difference over all pairs of policy trees is the maximum difference between V_t and V_{t-1}.

The linear program and pseudocode for this method appear in Table 8 and Table 9, respectively.

7.2.2 A weak method

The previous section described a method for computing the maximum difference between two piecewise-linear and convex value functions over belief space. This section describes a simpler approach that is much more efficient to compute but only gives a weak upper bound on the difference.

We'd like to find a bound on the biggest difference between two value functions, $X(b) = b \cdot \mathbf{value}(\mathbf{best}(b, \mathcal{X}))$ and $Y(b) = b \cdot \mathbf{value}(\mathbf{best}(b, \mathcal{Y}))$. The following theorem gives a cheap way to bound the biggest positive difference between X and Y.

Theorem 7 *(Weak value function comparison): Let*

$$\delta = \max_{p_x \in \mathcal{X}} \min_{p_y \in \mathcal{Y}} \max_{s \in \mathcal{S}} (\mathbf{value}(p_x)[s] - \mathbf{value}(p_y)[s]).$$

[3]In fact, the method we use is a simplification of the weak method described here.

Inputs:

$p_x, \mathcal{X}, p_y, \mathcal{Y}$

Variables:

$b[s]$ for each $s \in \mathcal{S}$

Maximize: $b \cdot (\mathbf{value}(p_x) - \mathbf{value}(p_y))$

Region constraints:

For each $\tilde{p_x}$ in $\mathcal{X}$: $b \cdot \mathbf{value}(p_x) \geq b \cdot \mathbf{value}(\tilde{p_x})$

For each $\tilde{p_y}$ in $\mathcal{Y}$: $b \cdot \mathbf{value}(p_y) \geq b \cdot \mathbf{value}(\tilde{p_y})$

Simplex constraints:

For each $s \in \mathcal{S}$: $b[s] \geq 0$

$\sum_{s \in \mathcal{S}} b[s] = 1$

Table 8: The linear program used to compute the maximum value difference between two policy trees.

```
difference(V_{t-1}, V_t) := {
    maxdiff := -∞
    foreach p_t in V_t {
        foreach p_{t-1} in V_{t-1} {
            LP := set up the difference LP (Table 8) with p_t, V_t, p_{t-1}, V_{t-1}
            (objective_1, b) := solveLP(LP)
            if (objective_1 = nil) objective_1 := -∞
            LP := set up the difference LP (Table 8) with p_{t-1}, V_{t-1}, p_t, V_t
            (objective_2, b) := solveLP(LP)
            if (objective_2 = nil) objective_2 := -∞
            maxdiff := max(maxdiff, objective_1, objective_2)
        }
    }
    return maxdiff
}
```

Table 9: Pseudocode for computing the Bellman residual exactly in polynomial time.

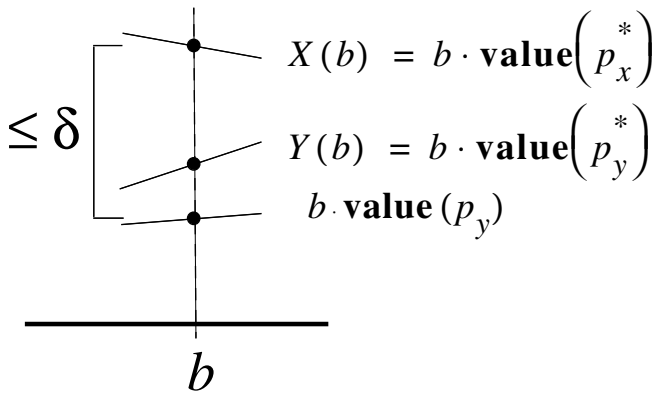

Figure 8: Weak bound on the maximum difference between value functions.

Then for all belief states, b, $X(b) - Y(b) \leq \delta$.

Proof: Consider some particular belief state, b. Let $p_x^* = \mathbf{best}(b, \mathcal{X})$, $p_y^* = \mathbf{best}(b, \mathcal{Y})$, and

$$p_y = \arg \min_{\tilde{p_y} \in \mathcal{Y}} \max_{s \in \mathcal{S}} (\mathbf{value}(p_x^*)[s] - \mathbf{value}(\tilde{p_y})[s]).$$

Further, define δ as in the statement of the theorem. This situation is depicted in Figure 8.

We can bound,

$$\begin{aligned}
X(b) - Y(b) &= b \cdot (\mathbf{value}(p_x^*) - \mathbf{value}(p_y^*)) \\
&\leq b \cdot (\mathbf{value}(p_x^*) - \mathbf{value}(p_y)) \\
&\leq \max_{s \in \mathcal{S}} (\mathbf{value}(p_x^*)[s] - \mathbf{value}(p_y)[s]) \\
&\leq \min_{\tilde{p_y} \in \mathcal{Y}} \max_{s \in \mathcal{S}} (\mathbf{value}(p_x^*)[s] - \mathbf{value}(\tilde{p_y})[s]) \\
&\leq \max_{\tilde{p_x} \in \mathcal{X}} \min_{\tilde{p_y} \in \mathcal{Y}} \max_{s \in \mathcal{S}} (\mathbf{value}(\tilde{p_x})[s] - \mathbf{value}(\tilde{p_y})[s]) \\
&\leq \delta,
\end{aligned}$$

as desired. Since b is arbitrary, the bound holds everywhere. □

Although this bound can be arbitrarily weak, it is a good approximation in the following sense. If the policy trees in $\mathcal{X}$ are identical to the policy trees in $\mathcal{Y}$, the weak bound will (correctly) state that the two value functions are

```
weakbound(𝒳, 𝒴) := {
    δ := −∞
    foreach p_x in 𝒳 {
        mindiff := ∞
        foreach p_y in 𝒴 {
            maxcomponent := max_{s∈𝒮}(value(p_x)[s] − value(p_y)[s])
            mindiff := min(mindiff, maxcomponent)
        }
        δ := max(δ, mindiff)
    }
    return δ
}
```

Table 10: Pseudocode for computing a bound on difference between two value functions.

```
difference'(𝒱_{t−1}, 𝒱_t) := {
    return max(weakbound(𝒱_{t−1}, 𝒱_t), weakbound(𝒱_t, 𝒱_{t−1}))
}
```

Table 11: Pseudocode for a weak bound on the Bellman residual.

identical. And, although it is difficult to formalize, if the two sets are only slightly different, the given bound will be fairly accurate.

Tables 10 and 11 provide pseudocode for the weaker bound.

7.3 Derivation of value

By definition, **value**$(p)[s]$ is the expected discounted reward for following t-step policy tree p starting from state s. Equation 12 gives a recursive formula for this expression. This section justifies that formula.

Let a be $\mathbf{action}(p)$. We have:

$$\begin{aligned}
\mathbf{value}(p)[s] &= R[s,a] + \gamma \sum_{s',o} \Pr(o,s'|a,s)\mathbf{value}(\mathbf{choice}(p,o))[s'] \\
&= R[s,a] + \gamma \sum_{s',o} \Pr(s'|a,s)\Pr(o|s,a,s')\mathbf{value}(\mathbf{choice}(p,o))[s'] \\
&= R[s,a] + \gamma \sum_{o,s'} T[s,a,s']O[s',a,o]\mathbf{value}(\mathbf{choice}(p,o))[s'] \\
&= R[s,a] + \gamma \sum_{o} \mathbf{back}(\mathbf{value}(\mathbf{choice}(p,o)),a,o)[s]
\end{aligned}$$

as desired.

7.4 Eliminating extraneous policy trees

Given a set $\mathcal{X}$ of policy trees, how can we find a minimum-sized subset, $\mathcal{V} \subseteq \mathcal{X}$, that represents the same value function?

Monahan [Monahan, 1982] describes a method that is further explored by Eagle [Eagle, 1984] for eliminating extraneous policy trees from a set. Interestingly, Monahan attributes the method to Sondik [Smallwood and Sondik, 1973] but the two methods are only similar at a high level. The method described here is a slight variant of the one discussed by Monahan.

The basic method is to consider each $p \in \mathcal{X}$ in turn and to ask, "Is there a belief state, b, such that the value achieved by following p starting from b is larger than that of following any other policy tree in $\mathcal{X}$?"

Once again, there is a simple linear program for answering this question. The linear program is given in Table 12.

The linear program searches for a b and a δ such that a given policy tree, p, gives a value of at least δ more than any other policy tree in the set $\mathcal{X}$ at b.

If the linear program returns a negative value for δ, then p must be dominated everywhere and therefore extraneous in the manner shown in Figure 6(a). If the returned value for δ is zero, that means that, at b, policy tree p is optimal but that there is no belief state for which policy tree p strictly dominates all the others. Thus p is again extraneous, but in the sense of Figure 6(b) or (c).

Only p's for which $\delta > 0$ are kept. These policy trees will not be extraneous since removing them from the set would change the value function being

Inputs:

$p, \mathcal{X}$

Variables:

δ, $b[s]$ for each $s \in \mathcal{S}$

Maximize: δ

Domination constraints:

For each $\tilde{p}$ in $\mathcal{X}$: $b \cdot \mathbf{value}(p) \geq \delta + b \cdot \mathbf{value}(\tilde{p})$

Simplex constraints:

For each $s \in \mathcal{S}$: $b[s] \geq 0$

$\sum_{s \in \mathcal{S}} b[s] = 1$

Table 12: A linear program used to decide whether policy tree p is extraneous in the set $\mathcal{X}$.

```
purge(X) := {
    V := {}
    foreach p in X {
        LP := set up the Monahan LP (Table 12) with p, X
        (δ, b) := solveLP(LP)
        if (δ ≤ 0) X := X − {p}
        else V := V ∪ {p}
    }
    return V
}
```

Table 13: Pseudocode for purging extraneous policy trees from a set.

represented.

The running time for this procedure is that of solving one linear program for each policy tree in $\mathcal{X}$. Each linear program has $|\mathcal{S}| + 1$ variables and at most one constraint for each policy tree in $\mathcal{X}$. Thus it runs in time polynomial in $|\mathcal{S}|$ and $|\mathcal{X}|$ (and the precision).

7.5 Ties in choosing the best vector

The most difficult part of the witness algorithm, both computationally and conceptually, is the identification of a witness point. Recall that given an approximation of Q_t^a, b is a witness if $Q_t^a(b) \neq \hat{Q}_t^a(b)$. In addition to finding a witness, b, we must select a policy tree, p, to add to $\hat{\mathcal{Q}}_t^a$ to improve the approximation. We need to find a p such that $\mathbf{value}(p) \cdot b = Q_t^a(b)$. But if there are multiple policy trees for which this is true, which do we add?

If we are very careless, we might add an extraneous policy tree such as p_2 shown in Figure 6(b) which contributes only a single, redundant point to the representation of Q_t^a. In this section, we argue that breaking ties between policy trees using a lexicographic ordering on their value vectors ensures that no extraneous policy trees are included. Further, we show how to identify the lexicographically best policy tree in the set of all optimal policy trees at b without having to enumerate this set.

7.5.1 Motivation

It is worth asking whether or not this is a significant problem. Consider the value function shown in Figure 9. It consists of 3 segments and, in the worst possible case, there are only 5 extraneous policy trees that might be included if we break ties arbitrarily. In general, the number of extraneous policy trees can not be greater than the number of intersection points between segments (or segments and the edges of belief space). Therefore, in two dimensions, this at most doubles the number of policy trees that are considered.

Since the extraneous policy trees can be eliminated quickly using the techniques in Section 7.4, this would seem to imply that the problem of ties contributes at most a constant factor to the computation. However, this is not true. The number of intersection points can blow up exponentially with the size of the state space, $|\mathcal{S}|$, and this would add significant overhead to the

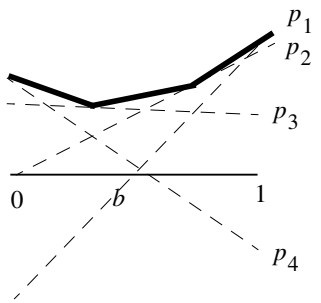

Figure 9: A value function and some extraneous vectors.

computation. It is important, at least for the analysis, that these extraneous policy trees not take over the computation time.

7.5.2 Lexicographic tie breaking

Let $\mathcal{U}$ be the set of vectors derived from all t-step policy trees,

$$\mathcal{U} = \{\mathbf{value}(p) : p \in \mathcal{P}_t\}.$$

Let $\mathcal{W}$ be the set of vectors corresponding to policy trees that are optimal at b,

$$\mathcal{W} = \{\mathbf{value}(p) : p \in \mathcal{P}_t \text{ and } \mathbf{value}(p) \cdot b = Q_t^a(b)\}.$$

This section argues that the *lexicographically largest*, vector in $\mathcal{W}$ corresponds to a policy tree that can not be extraneous. We say α_1 is lexicographically greater than α_2 ($\alpha_1 \succ \alpha_2$) if α_1's first component is bigger than α_2's first component. If they are tied at the first component, the tie is broken by the second component, and so on. The upshot of this is that any set of $|\mathcal{S}|$-vectors, such as $\mathcal{W}$, has exactly one lexicographic maximum. We define λ to be the lexicographic maximum vector in $\mathcal{W}$. That is, for all $\alpha \in \mathcal{W} - \{\lambda\}$, $\lambda \succ \alpha$.

Our goal is to show that the policy tree corresponding to λ cannot be extraneous. The proof of this is more complicated than it deserves to be. It is included below completeness. The intuition behind the proof is that we can move a little bit away from b, increasing its first component a bit, its second component a little bit less, and so on, so that λ gives an answer that is a tiny bit bigger than any of the other vectors in $\mathcal{U}$.

To show this formally, we need a little notation. Let e_s be the vector corresponding to the "corner" of belief space where all the probability mass is on state s. That is, $e_s[s] = 1$ and $e_s(\tilde{s}) = 0$, for $\tilde{s} \neq s$. The point $\tilde{b}$ is a *ρ-step from b_1 towards b_2* if $\tilde{b} = \rho(b_1 - b_2) + b_2$ since the expression evaluates to b_1 when ρ is 0 and b_2 when ρ is 1 and interpolates linearly for $0 < \rho < 1$.

Lemma 2 *(Safe corner moves): If $\mathcal{W}$ is the set of vectors in $\mathcal{U}$ that are maximal at b, then there is a value $\rho > 0$ such that for the point $\tilde{b}$ that is a ρ-step from b towards a corner of belief space, e_s, the vectors in $\mathcal{W}$ still give a bigger answer than all the other vectors.*

Proof: For any $\alpha \in \mathcal{W}$ and any $\beta \in \mathcal{U} - \mathcal{W}$, $\alpha \cdot b - \beta \cdot b > 0$. Call the difference Δ. Let $\tilde{b} = \rho(e_s - b) + b$. We can find a value for $\rho > 0$ such that α is bigger than β at $\tilde{b}$. The following statements are equivalent:

$$\begin{aligned}
\alpha \cdot \tilde{b} &> \beta \cdot \tilde{b} \\
\alpha \cdot (\rho(e_s - b) + b) - \beta \cdot (\rho(e_s - b) + b) &> 0 \\
(1 - \rho)(\alpha \cdot b - \beta \cdot b) + \rho(\alpha \cdot e_s - \beta \cdot e_s) &> 0 \\
(1 - \rho)\Delta + \rho(\alpha[s] - \beta[s]) &> 0 \\
\rho(\Delta + \beta[s] - \alpha[s]) &< \Delta
\end{aligned}$$

Since $\Delta > 0$, we can divide by it without changing the inequality. Let $\kappa = 1 + \frac{\beta[s] - \alpha[s]}{\Delta}$. Then the above expression is equivalent to $\rho\kappa < 1$.

If $\kappa \leq 0$, the inequality holds with $\rho = 1$. Otherwise, we can set $\rho = \frac{1}{2\kappa}$ to satisfy the inequality.

This shows that we can find a $\rho > 0$ for each $\alpha \in \mathcal{W}$ and $\beta \in \mathcal{U} - \mathcal{W}$ such that a ρ-step from b to e_s gives a $\tilde{b}$ such that $\alpha \cdot \tilde{b} > \beta \cdot \tilde{b}$ (that is, the set $\mathcal{W}$ still contains the winners). Since there are a finite number of ways of pairing elements in $\mathcal{W}$ with those in $\mathcal{U} - \mathcal{W}$, there is a $\rho > 0$ that works for *all* pairs (namely, the minimum ρ for any pair). □

We can now use the lemma to prove the following theorem.

Theorem 8 *(Lexicographic winner): If $\mathcal{W}$ is the set of vectors in $\mathcal{U}$ that are maximal at b, then there is some belief state, $\tilde{b}$, such that the lexicographically maximum vector, $\lambda \in \mathcal{W}$, is the sole winner at $\tilde{b}$. This means that the policy tree corresponding to λ is not extraneous.*

Proof: Let s_i be the state corresponding to the ith component of the vectors. Let $\mathcal{W}_0 = \mathcal{W}$ be the set of vectors maximal at $b_0 = b$. By Lemma 2, there is a point, b_1, strictly different from b_0 and in the direction of e_{s_1} where the vectors in $\mathcal{W}_0$ are still bigger than the others. A subset of these vectors, $\mathcal{W}_1$, are maximal at b_1.

It should be clear that the vectors in $\mathcal{W}_1$ are precisely those $\alpha \in \mathcal{W}_0$ for which $\alpha[s_1] = \lambda[s_1]$, that is, those tied with the lexicographically maximal vector in the first component.

If we continue to apply this argument for each component, $\mathcal{W}_i$ becomes the set of all vectors in $\mathcal{W}$ that agree with λ in the first i components. The vectors in $\mathcal{W}_i$ are strictly greater than the other vectors at b_i. After every component has been considered, we are left with $\mathcal{W}_{|\mathcal{S}|} = \{\lambda\}$ with λ as the sole winner at $b_{|\mathcal{S}|}$. □

This shows that the lexicographically maximum vector in the set of all tied vectors at b is guaranteed to be the sole winner at some point in belief space and therefore that its policy tree is not extraneous in the representation of V_t.

This tie breaking is needed at two places in the pseudocode given earlier. The first is in `best`, which finds the best policy tree in a set. Choosing the lexicographic maximum is easy in this case. The second place is in `besttree`, which returns the best policy tree given a point in belief space. It is not given the set of all policy trees in advance because this list could be exponentially long. Instead it constructs the tree by setting **choice** for each observation.

The correct choice for each observation is derived in Equation 13. It takes each policy tree in $\mathcal{V}_{t-1}$ and transforms it using **back**. It then chooses the policy tree whose transformed vector has a maximum dot product with the given belief state. This process is implemented in `besttree` (Table 3) which breaks ties between choices using the lexicographic maximum transformed vector.

We claim that breaking ties this way is guaranteed to generate a policy tree whose vector is lexicographically largest of all policy trees optimal at the given belief state. The argument is involved but the intuition is somewhat simple: Equation 12 shows that the vector for a policy tree is the componentwise sum of the immediate reward and the componentwise discounted components of the transformed choice vector. Since each choice vector is chosen to be the lexicographically largest of all the options, the sum will also be the lexicographically largest of all possible sums. Thus the generated policy

tree will have a vector which is the lexicographic maximum of all possible policy trees tied at the given belief state.

8 Contributions

This paper described a novel method for solving finite-horizon POMDP problems exactly using the belief state MDP. It repeated a classic analysis that shows that an infinite-horizon POMDP problem can be approximated arbitrarily well by a finite-horizon solution. It gave a new (to us) analysis that bounds the infinite-horizon performance of solving a sequence of finite-horizon problems approximately.

The paper repeated a result of Sondik [Smallwood and Sondik, 1973] that states that a finite-horizon value function is piecewise-linear and convex over belief space. It gave a novel proof of this that shows that we can interpret the components of the linear functions as the performance of a finite-horizon policy.

The paper showed how a t-step policy tree representation could be used to solve POMDP problems. It repeated a well-known result that we can find the optimal policy tree at a point in belief space and provided a novel analysis that showed that breaking ties using a lexicographic ordering ensures that the final policy tree representation will be of minimum size. It also described how lexicographic maximum vectors can be identified without forfeiting efficiency. In addition, two algorithms are given for finding the Bellman residual of a value function represented as a set of policy trees.

The most significant contribution is a philosophical one. The development of previous POMDP algorithms has been motivated by vague notions of algorithmic efficiency. This paper attempted to apply the theory of computational complexity to help identify promising algorithms and to help discard ones that will tend to take too long.

Acknowledgments

This is joint work with Tony Cassandra and Leslie Kaelbling (advisor). Loren Platzman provided several helpful background discussions. Philip Klein provided guidance on some of the more complicated arguments and pointed out

problems with the old algorithm. Ron Parr was kind enough to read an early draft front to back and provided many useful comments.

References

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

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

[Bertsekas, 1987] Bertsekas, D. P. 1987. *Dynamic Programming: Deterministic and Stochastic Models.* Prentice-Hall.

[Cassandra *et al.*, 1994] Cassandra, Anthony R.; Kaelbling, Leslie Pack; and Littman, Michael L. 1994. Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA.

[Cassandra, 1994] Cassandra, Anthony 1994. Optimal policies for partially observable Markov decision processes. Technical Report CS-94-14, Brown University, Department of Computer Science, Providence RI.

[Cheng, 1988] Cheng, Hsien-Te 1988. *Algorithms for Partially Observable Markov Decision Processes.* Ph.D. Dissertation, University of British Columbia, British Columbia, Canada.

[Drake, 1962] Drake, A. W. 1962. *Observation of a Markov Process Through a Noisy Channel.* Ph.D. Dissertation, Massachusetts Institute of Technology, Cambridge, Massachusetts.

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

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. 1979. *Computers and intractability: A guide to the theory of NP-completeness.* Freeman, San Francisco, CA.

[Heyman and Sobel, 1984] Heyman, D. and Sobel, M. 1984. *Stochastic Models in Operations Research: Stochastic Optimization*, volume 2. McGraw-Hill, New York.

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

[Platzman, 1977] Platzman, Loren K. 1977. *Finite-memory estimation and control of finite probabilistic systems.* Ph.D. Dissertation, Massachusetts Institute of Technology.

[Puterman, 1994] Puterman, Martin X. 1994. *Markov Decision Processes.* unknown.

[Schrijver, 1986] Schrijver, Alexander 1986. *Theory of linear and integer programming.* Wiley-Interscience.

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

[Sondik, 1971] Sondik, E. 1971. *The Optimal Control of Partially Observable Markov Processes.* Ph.D. Dissertation, Stanford University.

[Watkins, 1989] Watkins, C. J.C.H. 1989. *Learning with Delayed Rewards.* Ph.D. Dissertation, Cambridge University.

[Williams and Baird, 1993] Williams, Ronald J. and Baird, Leemon C. III 1993. Tight performance bounds on greedy policies based on imperfect value functions. Technical Report NU-CCS-93-13, Northeastern University, College of Computer Science, Boston, MA.