Efficient dynamic-programming updates in partially observable Markov decision processes

Michael L. Littman
Anthony R. Cassandra
Leslie Pack Kaelbling

December 12, 1995

Abstract

We examine the problem of performing exact dynamic-programming updates in partially observable Markov decision processes (POMDPs) from a computational complexity viewpoint. Dynamic-programming updates are a crucial operation in a wide range of POMDP solution methods and we find that it is intractable to perform these updates on piecewise-linear convex value functions for general POMDPs. We offer a new algorithm, called the witness algorithm, which can compute updated value functions efficiently on a restricted class of POMDPs in which the number of linear facets is not too great. We compare the witness algorithm to existing algorithms analytically and empirically and find that it is the fastest algorithm over a wide range of POMDP sizes.

1 Introduction

A partially observable Markov decision processes (POMDP) is a Markov decision process in which the decisions must be based solely on noisy and incomplete observations of the system's state. Although this model can be applied to a wider range of problems than can the Markov decision process model, it has received much less attention, in part because solving even the smallest POMDP is computationally demanding.

In this paper, we discuss the computational difficulty of making dynamic-programming updates in POMDPs and show that, if the class of POMDPs considered is restricted appropriately, a new approach called the witness algorithm can be used to perform these updates efficiently. We provide computational analyses of several prior algorithms for dynamic-programming updates in POMDPs, which show that each of these algorithms scales poorly in at least one critical task parameter. An empirical study of the algorithms supports these analyses and shows that the witness algorithm is able to solve POMDPs of a wider range of sizes; however, all exact solution methods are intractable for most large practical problems.

2 Problem Formulation

The reader is referred to the excellent surveys by Monahan [6], Lovejoy [5], and White [15] for a thorough introduction to solving POMDPs algorithmically. This section briefly presents notation and background results that enable us to define the problem addressed in this work. Our notation follows that of White.

A POMDP is defined by a system state space S, an observation space Z, and an action space A, all finite. A set of transition/observation matrices summarizes the dynamics of the model. We define $P(z, a)$ to be a matrix of probabilities; the i, jth component of $P(z, a)$, written $P(z, a)[i, j]$, is the probability that the system will make a transition to system state $j \in S$ and make observation $z \in Z$ given that action $a \in A$ was issued from system state $i \in S$. Immediate rewards for each system state under action a are given by the column vector $r(a)$.

In most POMDP applications, the objective is to find a *policy*, or decision rule, that selects actions so as to maximize the expected long-term reward. Since the system state is only partially observed, action decisions are based upon the history of observations and actions, which can be summarized in several ways. In this work, we summarize past history using probability distributions over system states, called *information states*, which are sufficient for making optimal decisions.

Let X be the set of information states and V^X the set of bounded, real-valued functions over X. A *value function* $v \in V^X$ assigns to each information state a measure of its overall utility. Of central importance to many POMDP algorithms is the *dynamic-programming update*. It can be expressed as an operator H that uses $P(z, a)$ and $r(a)$ to transform a value function v into a new value function Hv, which incorporates one additional step of rewards into v.

We will define H more carefully in the next section, but it is worth noting here its critical role in a host of POMDP solution methods. Dynamic-programming updates can be used for solving finite-horizon POMDPs [11], finding approximate solutions to infinite-horizon discounted POMDPs by value iteration [9], performing policy improvement in infinite-horizon algorithms [13], computing optimal average-reward policies [12], and accelerating convergence of successive approximation methods via reward revision [16]. Most of these approaches treat H as a primitive operation, so any improvement in the speed with which H can be computed immediately translates into speed-ups for a wide range of advanced POMDP algorithms.

Before we can state the problem addressed in this work, we need to commit ourselves to a specific representation for value functions. It is known that the dynamic-programming operator H preserves piecewise linearity and convexity and that piecewise-linear convex functions have a convenient representation as finite sets of vectors, described in more detail below. Therefore, if we represent a value function v as a set of vectors Γ, then Hv can be represented by a set of vectors Γ'. With this background established, we can define the central problem of this work.

Problem Statement *Given a set of vectors Γ representing a value function $v \in V^X$, compute a minimal set of vectors Γ' representing Hv.*

Although many algorithms for solving this problem have been proposed over the last twenty years [11, 6, 3, 2, 15], the algorithm described in this work is the first developed with close attention to issues of computational complexity and is currently the most efficient exact algorithm for performing dynamic-programming updates over a wide range of POMDP sizes.

We will now define the dynamic-programming operator H. Let $0 \leq \beta \leq 1$ be a discount factor. Let $H^a : V^X \rightarrow V^X$ be the dynamic-programming operator for action $a \in A$, specifically, if $v(x)$ is the value of information state x, then $[H^a v](x)$ is the expected immediate reward for taking action a plus the discounted expected value of the information state resulting from taking action a from information state x.

We can define the dynamic-programming operator H by

$$[Hv](x) = \max_a([H^a v](x)).$$

If $v(x)$ is the value of x, then $[Hv](x)$ is the expected immediate reward plus the discounted expected value of the information state corresponding to taking the best action from information state x.

Smallwood and Sondik [11] showed that H^a and H preserve piecewise linearity and convexity and therefore that Hv can be represented in a convenient form if v is piecewise linear and convex.

Theorem 1 *If $v \in V^X$ is a piecewise-linear convex function represented as a set of vectors Γ, such that*

$$v(x) = \max_{\gamma \in \Gamma}(x \cdot \gamma) ,$$

then there exist finite sets of vectors G^a and G, such that

$$[H^a v](x) = \max_{\gamma \in G^a}(x \cdot \gamma)$$

and $[Hv](x) = \max_{\gamma \in G}(x \cdot \gamma)$.

Smallwood and Sondik's proof gives a very explicit form for the sets G and G^a. Let $\mathcal{T}$ be the set of functions mapping observations to vectors in Γ. The sets G^a and G are defined as

$$G^a = \{[r(a) + \beta \sum_z P(z,a) \cdot g(z)] | g \in \mathcal{T}\}$$

and $G = \bigcup_a G^a$. We say a vector $\gamma \in G^a$ is *constructed from* $g \in \mathcal{T}$ if $\gamma = [r(a) + \beta \sum_z P(z,a) \cdot g(z)]$.

We define $\Gamma' \subseteq G$ and $\Gamma^a \subseteq G^a$ to be the sets of minimal size that represent Hv and $H^a v$, respectively. It is the case that Γ' and Γ^a are often a great deal smaller than G and G^a, since some of the vectors may give maximal dot products for no $x \in X$ (see Figure 1); for efficiency reasons, it is these minimal representations that the POMDP algorithms should manipulate.

To define Γ' and Γ^a in a way that makes them easier to compute, we need to introduce a few additional concepts. We define an arbitrary order on the system states in S and let γ be an $|S|$-vector with $\gamma[i]$ denoting the component of γ corresponding to system state $i \in S$. We say γ_1 is *lexicographically greater* than γ_2 if there is some system state $i \in S$ such that $\gamma_1[i] > \gamma_2[i]$ and $\gamma_1[i'] = \gamma_2[i']$ for all $i' < i$. We can also inductively define the lexicographic maximum vector in a finite set.

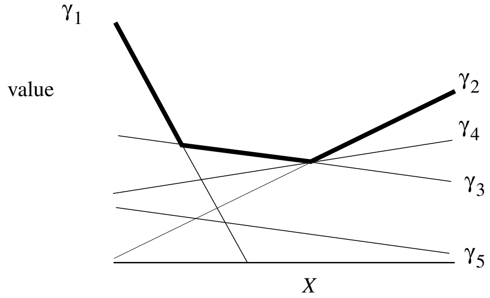


Figure 1: An illustration of a slice of a value function and the vectors that define it. Note that the vectors γ_4 and γ_5 can be eliminated without changing the value function.

Let F be a finite set of vectors. We say γ *dominates* all vectors in F at $x \in X$ if

$$x \cdot \gamma > x \cdot \gamma'$$

for all $\gamma' \in F - \{\gamma\}$. Note that γ may or may not be in F and that the defining inequality is strict. Let $R(\gamma, F)$ represent the set of all $x \in X$ for which the above inequality holds; that is, $R(\gamma, F)$ is the region of X over which γ dominates all other vectors in F. For convenience, we define $R(\gamma, \emptyset) = X$. We call a vector γ *useful* with respect to a set F if $R(\gamma, F)$ is non-empty, because removing γ from F would change the piecewise-linear convex function that F represents.

The following lemma, proved in Appendix A.1, links the notions of lexicographic maximum vectors and domination. It is important for guaranteeing that particular vectors in G^a are useful.

Lemma 1 *Given an information state x and a set of vectors F, let γ^* be the lexicographic maximum vector in F such that $x \cdot \gamma^* = \max_{\gamma \in F}(x \cdot \gamma)$. Then there exists an x' such that $x' \in R(\gamma^*, F)$, that is, γ^* dominates all other vectors in F at x'.*

We can now prove an important property of the representation of piecewise-linear convex functions by minimal sets of vectors. It shows that Γ^a and Γ' have a computationally convenient characterization with respect to G^a and G.

Theorem 2 *Let $v \in V^X$ be a piecewise-linear convex function over X such* *that* $v(x) = \max_{\gamma \in F}(x \cdot \gamma)$ *for some finite set of vectors* F*. Let*

$$\Phi = \{\gamma : \gamma \in F \text{ and } R(\gamma, F) \neq \emptyset\}.$$

Then the vectors in Φ *are necessary and sufficient for representing* v*; that is, for any strict subset* U *of* Φ *there exists an* $x \in X$ *such that* $v(x) \neq \max_{\gamma \in U}(x \cdot \gamma)$*, and at the same time,* $v(x) = \max_{\gamma \in \Phi}(x \cdot \gamma)$*.*

Proof: We first show that every vector $\gamma \in \Phi$ is necessary for representing v. Consider any $x \in R(\gamma, F)$. Such an x exists by construction of Φ. From the definition of R, we know that no other vector in F gives as large a dot product with x as γ does. Therefore, the value function represented by any set $U \subseteq F - \{\gamma\}$ would have a strictly smaller value at x than v does. Thus, every $\gamma \in \Phi$ is needed to represent v.

Next, we show that Φ is a sufficient representation of v, that is, $v(x) = \max_{\gamma \in \Phi}(x \cdot \gamma)$. To see this, consider any $x \in X$. Let γ^* be the lexicographic maximum vector in F such that $x \cdot \gamma^* = \max_{\gamma \in F}(x \cdot \gamma)$. By Lemma 1, there exists an x' such that $x' \in R(\gamma^*, F)$, so γ^* will be included in Φ. Thus, $v(x) = \max_{\gamma \in F}(x \cdot \gamma) = x \cdot \gamma^* = \max_{\gamma \in \Phi}(x \cdot \gamma)$. Since this holds for every x, Φ represents the same function that F does. □

The problem addressed in this work is therefore equivalent to finding the set

$$\Gamma' = \{\gamma : \gamma \in G \text{ and } R(\gamma, G) \neq \emptyset\}.$$

The next section explores what it means to compute Γ' efficiently.

3 Computational Complexity

This section discusses concepts from the theory of computational complexity as they apply to the problem of performing dynamic-programming updates in POMDPs, in particular, to the problem of computing Γ' from Γ. We show that the problem is likely to be intractable unless we restrict the class of POMDPs considered. We propose a particular class of POMDPs in which $\sum_a |\Gamma^a|$ is restricted, and show that problems in this class can be solved efficiently by the witness algorithm.

Intuitively, an algorithm is *efficient* if it can be used to solve reasonable-sized instances of a problem in a reasonable amount of time. In computer science, this has been formalized as follows. An efficient algorithm is one whose *running time* (a measure of the number of primitive operations required to solve a problem instance) grows more slowly than a polynomial function of the *input size* of the problem instance (a measure of the number of bits it would take to write down the problem instance). In the context of dynamic-programming updates in POMDPs, we have the following definition of efficiency.

Definition *An algorithm for computing Γ' is* efficient *if its worst-case running time can be bounded above by some polynomial function of $|S|$, $|Z|$, $|A|$, and $|\Gamma|$.*

If the running time of an algorithm grows faster than any polynomial function of the input size, there is little hope of using it to solve large problem instances. Unfortunately, we can show that no algorithm can compute Γ' from Γ efficiently for general POMDPs, simply because Γ' can be exponentially large with respect to the size of the POMDP.

Theorem 3 *There exists a family of* POMDP*s such that, for every n, $|S| = 2n$, $|A| = 1$, $|Z| = n$, $|\Gamma| = 2$, and $|\Gamma'| = 2^n$.*

A proof by construction can be found in Appendix A.2.1.

To produce an efficient algorithm for computing Γ', we first must rule out the possibility that Γ' is exponentially large. We can dismiss this possibility altogether by restricting our attention to POMDPs for which $|\Gamma'|$ is polynomially bounded. We call a family of POMDPs *polynomially output bounded* if $|\Gamma'|$ can be bounded by a polynomial in the size of the POMDP.

No existing algorithm has been shown to run in polynomial time on polynomially output-bounded POMDPs, and the next theorem suggests that there may be a good reason for this.

Theorem 4 *The best algorithm for solving polynomially output-bounded* POMDP*s is efficient if and only if $\mathcal{RP} = \mathcal{NP}$.*

The theorem is proved in Appendix A.3. Its importance is that it links the problem of exactly solving POMDPs with the question of whether $\mathcal{RP} = \mathcal{NP}$, one of the fundamental open problems in theoretical computer science. This theorem says that a proof that a given algorithm solves polynomially output-bounded POMDPs efficiently would constitute a proof that every combinatorial problem in a wide and much studied class ($\mathcal{NP}$) has an efficient randomized algorithm. As the question of the existence of efficient algorithms for problems in $\mathcal{NP}$ has been open for some time, Theorem 4 can be taken as strong evidence that the problem of solving general polynomially output-bounded POMDPs is intractable.

We conclude from these results that we must further restrict the class of POMDPs under consideration if we hope to develop an efficient algorithm. We say a family of POMDPs is *polynomially action-output bounded* if $\sum_{a \in A} |\Gamma^a|$ is bounded by a polynomial in the size of the POMDP.

The quantity $\sum_{a \in A} |\Gamma^a|$ is an upper bound on $|\Gamma'|$, though the bound may be arbitrarily loose. By focusing on polynomially action-output-bounded POMDPs, we can solve for Γ' in polynomial time as long as we can find Γ^a in polynomial time for each $a \in A$. We therefore restrict our attention to polynomially action-output-bounded POMDPs, although other restrictions might also be useful.

Even in this restricted class of POMDPs, existing algorithms for computing Γ' are not efficient. Theorem 5 addresses the worst-case running times for the two algorithms studied in Section 5 and shows that they are not efficient. In Section 4, we will show that the witness algorithm can solve polynomially action-output-bounded POMDPs in polynomial time.

Theorem 5 *Any algorithm for solving polynomially action-output-bounded* POMDP*s that enumerates the vectors in G or the vertices of the linear regions in Hv takes exponential time in the worst case.*

Proof: Several algorithms for finding Γ' work by enumerating the vectors in G and then identifying which of these vectors is useful: Monahan's algorithm [6] was the first and later Eagle [3] and Lark [15] provided improvements. However, all these algorithms, regardless of their details, build G, the size of which is $|A||\Gamma|^{|Z|}$. Thus, even if a vector could be identified as useful in constant time, the running times of these algorithms are at least exponential in $|Z|$, making them of little use for solving POMDPs with anything but the smallest observation sets.

Cheng's linear support and relaxed region algorithms [2] make use of special-purpose routines that enumerate the vertices of each linear region of the value function. For polynomially action-output-bounded POMDPs, the number of linear regions is guaranteed to be small, but the number of vertices of each region can be quite large. Bounding the number of vertices in a polyhedron is a well-studied problem and it is known that there can be an exponential number. In fact, there is a family of POMDPs such that, for every n, $|S| = n+1$, $|A| = 2n+1$, $|Z| = 1$, $|\Gamma| = 1$, $|\Gamma'| \leq 2n+1$, and the number of vertices in one of the regions is 2^n. The construction is given in Appendix A.2.2. Since visiting each vertex is just one of the operations the algorithms perform, we can expect the worst-case running time to grow at least exponentially in n. □

In Section 5, we measure the running time for several of these algorithms on randomly-generated POMDPs and find that the worst-case analyses qualitatively match the average-case running times.

4 The Witness Algorithm

This section describes the witness algorithm for dynamic-programming updates in POMDPs, justifies it theoretically, and analyzes its computational complexity. The discussion proceeds top down.

At the highest level, the witness algorithm implements Hv by computing Γ^a for each $a \in A$ and creating the set

$$\Gamma' = \left\{ \gamma | \gamma \in \bigcup_a \Gamma^a \text{ and } R(\gamma, \bigcup_a \Gamma^a) \neq \emptyset \right\}.$$

It follows from Theorem 2 that Γ' is properly defined. The algorithm is given in Table 1.

Two subroutines still need to be defined. The subroutine `inR`(γ, F) returns an information state $x \in R(\gamma, F)$, that is, an x such that γ dominates all other vectors in F at x. If $R(\gamma, F)$ is empty, `inR` returns the token "`false`." The subroutine `H`$^{\mathtt{a}}(\Gamma)$ computes Γ^a from Γ.

Table 2 provides a simple routine that uses linear programming to implement `inR` efficiently. The linear program seeks an $x \in X$ such that the minimum increment of γ over the other vectors in F is as large as possible. If this increment is strictly greater than zero, then γ is useful with respect to F and x is in $R(\gamma, F)$. Note that if we assume that all the components of γ and the vectors in F are specified using rational numbers, the linear program can be implemented in such a way that the comparison to zero is well defined.

The witness approach to computing Γ^a bears a resemblance to Cheng's linear support algorithm [2] and Lark's algorithm [15]. The fundamental difference is that it does not exhaustively enumerate vectors or vertices. It

```
H(Γ) := {
    foreach a ∈ A
        Γ^a := H^a(Γ)
    return reduce(⋃_{a∈A} Γ^a)
}

reduce(F) := {
    U := ∅
    foreach γ ∈ F
        if (inR(γ, F) ≠ false) then add γ to U
    return U
}
```

Table 1: The witness approach to computing H.

```
inR(γ, F) := {
    if (F = ∅) then return any x ∈ X
    Solve the following linear program:
        maximize: d
        s.t.: x · γ − x · γ' ≥ d, for all γ' ∈ F − {γ}
        and: Σ_i x[i] = 1
        variables: x[i], d (nonnegative)
    if (d > 0) then return x
    else return false
}
```

Table 2: Subroutine for finding an information state in $R(\gamma, F)$.

starts with a set U, initially empty, and brings the vectors of Γ^a, one by one, into U, until $U = \Gamma^a$. The steps are as follows:

1. If we can verify that $U = \Gamma^a$, stop and return U.

2. Choose a vector $\gamma^+ \in G^a - U$. If γ^+ is useful with respect to U, we can use it to identify a vector in $\Gamma^a - U$.

3. If $R(\gamma^+, U) = \emptyset$, return to step 1. Otherwise let $x \in R(\gamma^+, U)$.

4. Let γ^* be the lexicographically maximum vector in G^a such that $x \cdot \gamma^* = \max_{\gamma' \in G^a}(x \cdot \gamma')$. Add γ^* to U. Return to step 1.

The justification for step 4 comes from Lemma 1, which shows that $\gamma^* \in \Gamma^a$. Step 3 can be implemented using the `inR` subroutine. Care must be taken in steps 1 and 2, described below, to make sure that the number of γ^+ vectors tested grows with Γ^a, not G^a. Implementing step 4 directly is similarly worrisome since it appears to require an enumeration of G^a. The following theorem shows that this is not necessary.

Theorem 6 *Let $x \in X$ be an information state and $a \in A$ an action. Let $g \in \mathcal{T}$ and define $g(z) = \arg\max_{\gamma \in \Gamma}(x \cdot P(z,a) \cdot \gamma)$ where ties in the "$\arg\max$" are broken in favor of the γ such that $P(z,a) \cdot \gamma$ is lexicographically largest. The vector $\gamma^* = [r(a) + \beta \sum_z P(z,a) \cdot g(z)]$ is useful in the representation of $H^a v$ and thus $\gamma^* \in \Gamma^a$.*

Proof: Starting with Theorem 1 we have

$$\begin{aligned}
[H^a v](x) &= \max_{\gamma \in G^a}(x \cdot \gamma) \\
&= \max_{g' \in \mathcal{T}} \left(x \cdot [r(a) + \beta \sum_z P(z,a) \cdot g'(z)] \right) \\
&= x \cdot r(a) + \beta \sum_z \max_{\gamma \in \Gamma}(x \cdot P(z,a) \cdot \gamma) \\
&= x \cdot [r(a) + \beta \sum_z P(z,a) \cdot g(z)] \\
&= x \cdot \gamma^*.
\end{aligned}$$

Since g is chosen so that $P(z,a) \cdot g(z)$ is lexicographically maximized for all z, no other element of $\mathcal{T}$ can be used to construct a lexicographically

```
useful-g-from-state(a, x, Γ) := {
    foreach z ∈ Z {
        maxval := −∞
        foreach γ ∈ Γ {
            if ((x · P(z, a) · γ > maxval) or
                    ((x · P(z, a) · γ = maxval)
                     and (lexgt(P(z, a) · γ, P(z, a) · g(z)))) then {
                maxval := x · P(z, a) · γ
                g(z) := γ
            }
        }
    }
    return g
}
```

Table 3: Subroutine for finding a $g \in \mathcal{T}$ that can be used to construct a useful vector at x.

larger vector while still satisfying the above constraint. By Lemma 1 and Theorem 2, such a vector is guaranteed to be useful. □

Table 3 provides an implementation of this idea. It uses the subroutine `lexgt`(ϕ, γ), which returns `true` if ϕ is lexicographically greater than γ.

Finally, we describe a method for limiting the set of vectors that are selected in step 2 in such a way that the test in step 1 can be carried out efficiently.

To do this, we first need to introduce a few additional concepts. We call the elements of $\mathcal{T}$ *tables* and say that table $g_1 \in \mathcal{T}$ is a *neighbor* of table $g_2 \in \mathcal{T}$ if $g_1(z) = g_2(z)$ for all but one $z \in Z$. Each table has $|Z|(|\Gamma| - 1)$ neighbors, which can be enumerated easily.

The next theorem says we only need to examine the neighbors of the tables that were used to construct the vectors in U to determine whether $U = \Gamma^a$. Further, if $U \neq \Gamma^a$, we can use one of the neighboring tables to find a vector $\gamma^+ \notin U$ such that $R(\gamma^+, U)$ is non-empty, as required in step 3.

Theorem 7 *Let $U \subseteq \Gamma^a$ be a non-empty set of vectors and assume each $\gamma \in U$ was constructed from $g^\gamma \in \mathcal{T}$, that is, $\gamma = r(a) + \beta \sum_z P(z, a) \cdot g^\gamma(z)$.*

Then $U \neq \Gamma^a$ if and only if there is some $g \in \mathcal{T}$ that is a neighbor of g^γ for some $\gamma \in U$ such that for $\gamma^+ = [r(a) + \beta \sum_z P(z,a) \cdot g(z)]$, $R(\gamma^+, U)$ is non-empty.

Proof: The "if" direction is easy since the g can be used to identify a vector missing from U.

The "only if" direction can be rephrased as: If $U \neq \Gamma^a$ then there is an information state $x \in X$, a $\gamma \in U$ constructed from g^γ, and a neighbor g of g^γ such that the vector constructed from g dominates all vectors in U at x. Figure 2 illustrates some of the relevant quantities used to show this.

Start by picking $\gamma^* \in \Gamma^a - U$ and choose any $x \in R(\gamma^*, U)$. Let

$$\gamma = \arg\max_{\gamma' \in U}(x \cdot \gamma').$$

As illustrated in the figure, γ^* is the vector in $\Gamma^a - U$ that gives the largest dot product with x, and γ is the vector in U that gives the largest dot product with x. By construction, $x \cdot \gamma^* > x \cdot \gamma$.

Now choose $g^* \in \mathcal{T}$ such that $\gamma^* = [r(a) + \beta \sum_z P(z,a) \cdot g^*(z)]$. As indicated in the figure, γ^* could be constructed from g^*, and γ was constructed from g^γ. If g^* and g^γ are neighbors, we are done, since we are searching for a neighbor of g^γ that can be used to construct a vector that dominates the other vectors in U at x, and g^* meets these requirements.

If g^* and g^γ are not neighbors, we will identify a $g \in \mathcal{T}$ that does satisfy these requirements. Choose an observation $z^* \in Z$ such that

$$x \cdot P(z^*, a) \cdot g^*(z^*) > x \cdot P(z^*, a) \cdot g^\gamma(z^*) \quad .$$

There must be a z^* satisfying this inequality since otherwise we get the contradiction

$$\begin{aligned} x \cdot \gamma^* &= x \cdot [r(a) + \beta \sum_z P(z,a) \cdot g^*(z)] \\ &\leq x \cdot [r(a) + \beta \sum_z P(z,a) \cdot g^\gamma(z)] = x \cdot \gamma \quad . \end{aligned}$$

Define $g(z^*) = g^*(z^*)$ and $g(z) = g^\gamma(z)$ for $z \neq z^*$. Let γ^+ be the vector constructed from g,

$$\gamma^+ = [r(a) + \beta \sum_z P(z,a) \cdot g(z)].$$

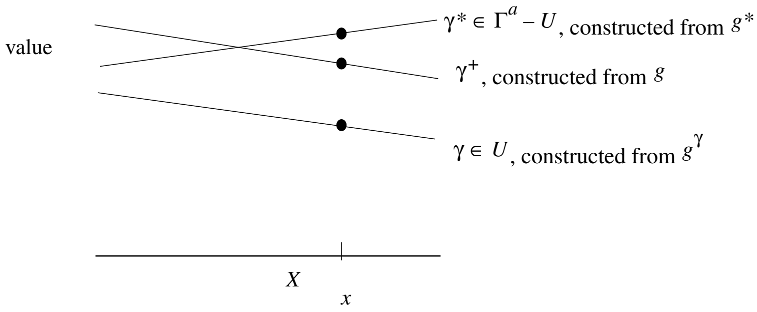


Figure 2: An illustration of some of the quantities used in Theorem 7.

Because $g(z) = g^\gamma(z)$ for all but one value of z, g is a neighbor of g^γ. In addition,

$$\begin{aligned} x \cdot \gamma^+ &= x \cdot [r(a) + \beta \sum_z P(z,a) \cdot g(z)] \\ &= x \cdot \left[r(a) + \beta \left(\sum_{z \neq z^*} P(z,a) \cdot g^\gamma(z) + P(z^*,a) \cdot g^*(z^*) \right) \right] \\ &> x \cdot [r(a) + \beta \sum_z P(z,a) \cdot g^\gamma(z)] = x \cdot \gamma = \max_{\gamma' \in U}(x \cdot \gamma'). \end{aligned}$$

Therefore the vector γ^+ constructed from g dominates all other vectors in U at x. □

Once γ^+ is identified, it serves as proof that U can be improved upon at the information state x. We can then use `useful-g-from-state` to construct a vector $\gamma^* \in \Gamma^a$ to include in U.

Table 4 provides a procedure that uses the insights discussed above to compute Γ^a from Γ; its fundamental steps are

- maintain a set `unchecked` of tables that might lead to useful vectors,
- construct the vector γ^+ for some table g from `unchecked`,
- find an information state x, if any, where γ^+ dominates all the vectors in U,
- find the best table, g^*, for x,
- construct the vector γ^* for g^*,

```
Hª(Γ) := {
    U := ∅
    unchecked := {any g ∈ 𝒯}
    while (unchecked ≠ ∅) {
        g := any element of unchecked
        γ⁺ := [r(a) + β Σ_z P(z, a) · g(z)]
        x := inR(γ⁺, U)
        if (x ≠ false) then {
            g* := useful-g-from-state(a, x, Γ)
            γ* := [r(a) + β Σ_z P(z, a) · g*(z)]
            add γ* to U
            add neighbors of g* to unchecked
        }
        else remove g from unchecked
    }
    return U
}
```

Table 4: The witness approach to computing Γ^a.

- add γ^* to U and the neighbors of g^* to `unchecked` and repeat.

Because of Theorem 7, the set `unchecked` need hold only the neighbors of the useful tables discovered thus far. The following theorem shows that the resulting algorithm is efficient.

Theorem 8 *The witness algorithm runs in polynomial time on polynomially action-output-bounded* POMDP*s.*

Proof: In computing Γ^a, the total number of tables added to `unchecked` is equal to the number of neighbors of the tables used to construct the vectors in Γ^a plus the arbitrarily chosen starting table, specifically, $1 + |Z|(|\Gamma| - 1)|\Gamma^a|$. Each pass through the "while" loop either consumes an element from `unchecked` ($1 + |Z|(|\Gamma| - 1)|\Gamma^a|$ times) or adds a vector to U ($|\Gamma^a|$ times). Thus, the total number of iterations in $\mathtt{H^a}$ is

$$1 + |Z|(|\Gamma| - 1)|\Gamma^a| + |\Gamma^a|.$$

The statements in the loop in $\mathtt{H^a}$ can all be implemented to run in polynomial time; this includes `inR`, since polynomial-time algorithms for linear programming with polynomial-precision rational numbers exist [10]. The total running time of $\mathtt{H^a}$ is therefore bounded by a polynomial in $|S|$, $|Z|$, $|A|$, $|\Gamma|$ and $|\Gamma^a|$.

The `H` routine calls $\mathtt{H^a}$ for each $a \in A$ and then calls `inR` once for each vector found. For polynomially action-output-bounded POMDPs, this implies that the total running time is polynomial. □

5 Empirical Results

The theoretical analyses leave open the issue of average-case performance and, for the witness algorithm, this includes the relationship between $\sum_a |\Gamma^a|$ and $|\Gamma'|$. In particular, are typical POMDPs polynomially action-output bounded? Also, the analyses ignore low-order terms, which might dominate the running time for small POMDPs.

In this section we empirically compare the witness algorithm to Cheng's linear support algorithm [2, 5] and Lark's enumeration algorithm [15]. We chose the linear support algorithm because it has been shown to outperform Sondik's one-pass algorithm and Cheng's relaxed region algorithm on a variety of POMDPs [2] and appears to be the leading exact algorithm. We chose Lark's enumeration algorithm because it seems to be the leading enumeration algorithm; in principle it should always outperform Monahan's algorithm, and did so in all our early experimental trials.

Our comparisons use CPU time as the measure of performance. Computer architectures, languages, operating systems, programming skill and other factors can combine to mask important algorithmic differences between different implementations. In an effort to minimize these effects, we used identical implementations for common subroutines wherever possible, ran all experiments on the same computer system, and took care to ensure there were no obvious inefficiencies in the programs. All experiments were performed on Sun Sparcstation 10 workstations running code written in the C language for the Solaris operating system. The code was compiled using the GNU C compiler (no optimization) with library calls to the CPLEX linear programming package.[1]

[1]We include implementation details for the sake of completeness and replicability. No recommendation of a particular software package or computer system is implied or

Another practical concern is the precision of the machine arithmetic. In all programs, we use the same tolerance of $\pm 10^{-9}$ for equality comparisons, such as the comparison with zero in `inR`. Though this does not guarantee that each algorithm will produce answers with exactly the same precision, none of the algorithms seemed to suffer from precision problems in the experiments reported here.

We will show how the three algorithms perform as we vary the number of system states and observations. Our preliminary experiments showed that, for the sizes of POMDPs we examined, the number of actions and previous vectors do not influence the running times of these algorithms nearly as much as do the number of system states and observations. The size of Γ' is not directly controllable, because it depends upon all of the input parameters. Instead of restricting our experiments to polynomially output-bounded POMDPs, we chose to view the size of Γ' as part of the POMDP instance itself; the results reported are averages over experiments with varying sizes of Γ'.

We tested the algorithms on randomly generated POMDPs because there are very few standard POMDPs in the literature and because we wanted to explore how the running time of the algorithms scaled with POMDP size. Additionally, using randomly generated POMDPs allowed us to run the algorithms on many different POMDPs of the same size, which helped smooth out the effect of the varying size of Γ'. It is certainly possible that the kinds of POMDPs that are of practical interest will exhibit different empirical complexity than these randomly-generated ones, perhaps due to structure in the transition matrices or properties of the immediate rewards. This topic deserves more attention.

Random POMDPs were constructed by independently generating random probability distributions for each row of the transition and observation matrices. A random distribution was generated by selecting a point uniformly at random from the $(|S|-1)$-dimensional simplex.[2] The immediate rewards for each action-state pair were set uniformly at random to values between 0.0 and 10.0.

In our experiments we kept the number of actions and the number of initial vectors fixed, $|A| = 4$, and $|\Gamma| = 10$. The latter were constructed by adding randomly generated vectors to the set until there were ten useful vectors. Our experiments consisted of performing a single dynamic-

intended.

[2]We thank John Hughes for pointing out how to do this.

programming update on each POMDP.

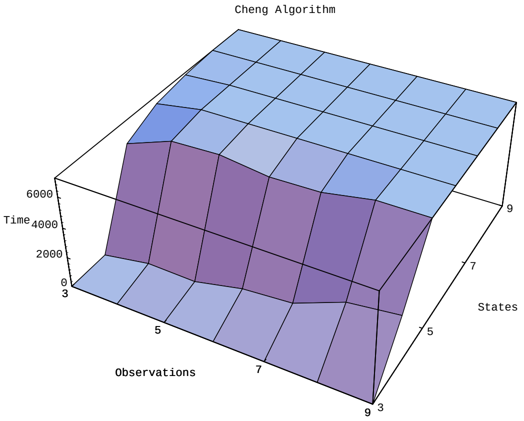
Our data are presented as a three-dimensional surface, where the height of the surface is the average computation time of the algorithm, in seconds. Figure 3 shows the results; running time is plotted against the number of system states and the number of observations. Each data point for a POMDP size is the average running time of the algorithm on 20 independent random POMDPs of that size. An experiment was terminated if it took more than 7200 seconds (2 hours) and that value was used as the computation time when computing an average.

These average-case empirical results match the theoretical worst-case analysis extremely well at a qualitative level. The performance of the linear support algorithm deteriorates rapidly as the number of system states increases, nearly independent of the number of observations. The enumeration algorithm suffers the same problem, except the deterioration increases dramatically with observations instead of the system states. Only the witness algorithm consistently found solutions over the entire range of POMDP sizes we tested.

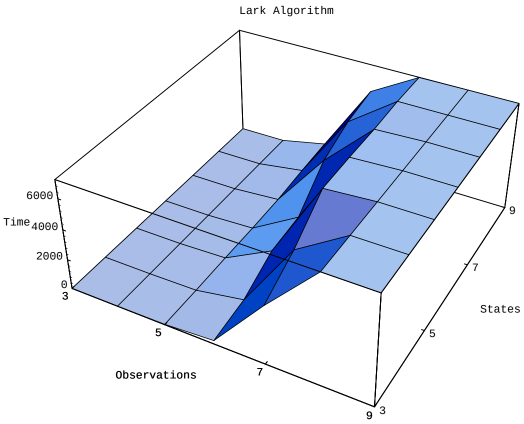
It is important to note that this data does not support the use of the witness algorithm for all POMDP instances. When there are fewer than four system states, the linear support algorithm appears to be the superior technique and when there are fewer than six observations, Lark's enumeration algorithm appears fastest. However, outside these ranges, our experiments indicate that the witness algorithm is likely to give the best performance.

The scale of the graph in Figure 3(c) does not provide a comprehensive view of the shape of the performance graph for the witness algorithm, since the saturation point of 7200 seconds has not yet been reached. Figure 4 shows the average running times over a larger range of system states and observations, where each data point is the average of 9 independent experiments. The results indicate that, as long as $|S| + |Z| < 22$, the witness algorithm is able to provide results within the allotted time.

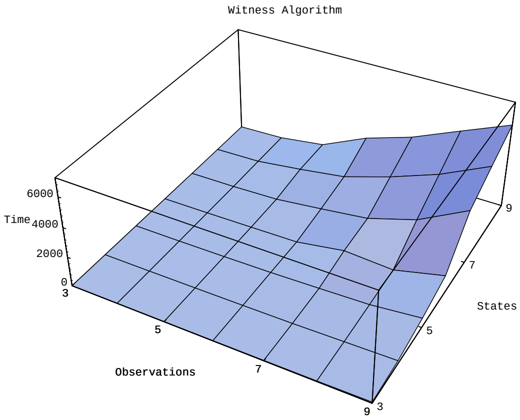
The theoretical analyses show that the the witness algorithm is efficient for computing the Γ^a sets, but can be inefficient for computing Γ'. Therefore, it is of interest to explore the relationship between $\sum_a |\Gamma^a|$ and $|\Gamma'|$. Using the same experiments as were used for the data of Figure 3, we collected statistics on the ratio between $\sum_a |\Gamma^a|$ and $|\Gamma'|$. We found the mean to be 2.65 with a variance of 3.74, giving some support to the idea that randomly generated POMDPs are polynomially action-output bounded. An important open question is whether this is true of typical POMDPs.



(a) Cheng



(b) Lark



(c) Witness

Figure 3: Running times of the three algorithms over a range of POMDP sizes.

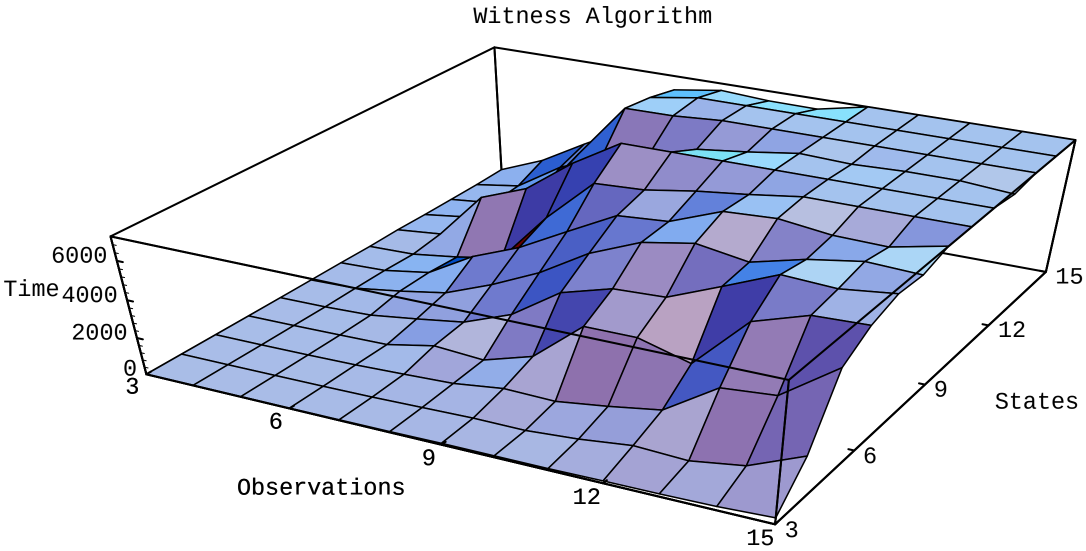


Figure 4: Running times of the witness algorithm over a larger range of POMDP sizes.

6 Conclusions

Attention to complexity theory has led us to a more practical approach to performing dynamic-programming updates in POMDPs with piecewise-linear convex value functions. The algorithm itself is almost as easy to implement as Monahan's algorithm, requiring only basic matrix manipulation and a standard linear-programming solver. The algorithm outperformed the best known algorithms on a suite of randomly generated POMDPs of various sizes. Although the existing implementation is reasonably efficient, improved implementations that take advantage of sparsity in $P(z, a)$ and that halt linear programming when a feasible solution is found would undoubtedly improve performance substantially.

We have applied our algorithm in the context of a value-iteration procedure for approximating optimal infinite-horizon policies. We have solved problems in a variety of domains, taken from the operations research, artificial intelligence, and machine learning literatures. Although it is known that solving multi-stage POMDPs is computationally intractable [7], we have found that we are often able to compute exact answers or reasonable approximations to larger POMDPs than have been reported elsewhere [1, 4].

Nevertheless, these exact algorithms do not appear to be practical for moderately large problems, and even small problems can exhibit explosive growth of the number of vectors in successive iterations. Our current work tries to overcome these difficulties using approximate representations of the infinite-horizon value function [4, 8].

A Appendix

A.1 Proof of Lemma 1

Given an information state x and a set of vectors F, let γ^* be the lexicographic maximum vector in F such that $x \cdot \gamma^* = \max_{\gamma \in F}(x \cdot \gamma)$. Lemma 1 asserts the existence of an x' such that $x' \in R(\gamma^*, F)$; that is, γ^* dominates all other vectors in F at x'.

To show this formally, we need some additional notation. Information state x is a *ρ-step from x_1 towards x_2* if $x = \rho(x_2 - x_1) + x_1$, $0 \leq \rho \leq 1$. Let e_i be the vector corresponding to the "corner" of X where all the probability mass is on system state $i \in S$.

Lemma 2 *Let $W = \{\gamma \in F | x \cdot \gamma \geq x \cdot \gamma' \text{ for all } \gamma' \in F\}$; that is, W is the set of vectors in F that dominate those in $F - W$ at x. For all $i \in S$, if $x \neq e_i$, there is a value $\rho > 0$ such that for the information state x' that is a ρ-step from x towards e_i, the vectors in W still dominate those in $F - W$.*

Proof: For any $\gamma \in W$ and any $\gamma' \in F - W$, define $\Delta = x \cdot \gamma - x \cdot \gamma' > 0$. Let $x' = \rho(e_i - x) + x$. We can find a value for $\rho > 0$ such that γ is bigger than γ' at x'. The following statements are equivalent:

$$\begin{aligned} x' \cdot \gamma &> x' \cdot \gamma' \\ (\rho(e_i - x) + x) \cdot \gamma - (\rho(e_i - x) + x) \cdot \gamma' &> 0 \\ (1 - \rho)(x \cdot \gamma - x \cdot \gamma') + \rho(e_i \cdot \gamma - e_i \cdot \gamma') &> 0 \\ (1 - \rho)\Delta + \rho(\gamma[i] - \gamma'[i]) &> 0 \\ \rho(\Delta + \gamma'[i] - \gamma[i]) &< \Delta \end{aligned}$$

Since $\Delta > 0$, we can divide by it without changing the inequality. Let $\kappa = 1 + (\gamma'[i] - \gamma[i])/\Delta$. Then the above expression is equivalent to $\rho\kappa < 1$. If $\kappa \leq 0$, the inequality holds with $\rho \leq 1$. Otherwise, we can set $\rho \leq 1/(2\kappa)$ to satisfy the inequality.

This shows that for every i, we can find a $\rho > 0$ for each pairing of $\gamma \in W$ and $\gamma' \in F - W$ such that a ρ-step from x towards e_i gives an x' such that $x' \cdot \gamma > x' \cdot \gamma'$. Since there are a finite number of ways of pairing elements in W with those in $F - W$, there is a $\rho > 0$ that works for *all* pairs (namely, the minimum ρ for any pair). $\square$

We can now use Lemma 2 to prove Lemma 1. The plan is to take successive ρ-steps from x towards each corner e_i. Each step is small enough so that the lexicographic maximum vector in W still dominates the vectors in $F - W$, but large enough so that some ties within W are broken.

Let $W_0 = W$ be the set of vectors maximal at $x_0 = x$. By Lemma 2, there is an information state, x_1, strictly different from x_0 if $x_0 \neq e_0$ and in the direction of e_0 at which the vectors in W_0 are still bigger than the others. Let W_1 be a subset of W_0 consisting of the vectors maximal at x_1. If $x_0 = e_0$, let $W_1 = W_0$.

It should be clear that the vectors in W_1 are precisely those $\gamma \in W_0$ for which $\gamma[0] = \gamma^*[0]$, that is, those tied with the lexicographically maximal vector in the first component.

If we apply this argument inductively for each component, W_i becomes the set of all vectors in W that agree with γ^* in the first i components. The vectors in W_i dominate the vectors in $F - W_i$ at x_i. After every component has been considered, we are left with $W_{|S|} = \{\gamma^*\}$ with γ^* as the unique vector dominating all others in F at $x_{|S|}$.

A.2 Constructed POMDPs

This section provides POMDP constructions that illustrate various important POMDP issues. We first define an alternate representation for piecewise-linear convex functions, prove that any function represented in this way is also the solution to a POMDP of similar size, and finally use the alternate representation to show the existence of POMDPs with particular properties.

Recall that to specify a POMDP dynamic-programming-update problem, it is necessary to define sets S, Z, and A; matrices $P(z, a)$ for $z \in Z$ and $a \in A$; vectors $r(a)$ for $a \in A$; a scalar β; and a set Γ of $|S|$-vectors. The set Γ' is the minimum set of vectors such that

$$\max_{\gamma \in \Gamma'}(x \cdot \gamma) = \max_{a \in A} \max_{g \in \mathcal{T}}(x \cdot [r(a) + \beta \sum_z P(z, a) \cdot g(z)]), \qquad (1)$$

for all $x \in X$, where $\mathcal{T}$ is the set of all functions mapping Z to Γ. We write

$v(x) = \max_{\gamma \in \Gamma}(x \cdot \gamma)$ and $[Hv](x) = \max_{\gamma \in \Gamma'}(x \cdot \gamma)$. The Z, A, P, r, β, and Γ quantities define the set Γ', and so can be viewed as a specification of a piecewise-linear convex function over the $(|S|-1)$-dimensional simplex.

We would like to be able to show that there are values for Z, A, P, r, β, and Γ such that piecewise-linear convex functions with particular qualities exist. Because POMDP specifications can be cumbersome, we restrict ourselves to a subset of the piecewise-linear convex functions resulting from dynamic-programming updates in POMDPs. We show that by specifying a finite set of $|S|$-vectors K, and a finite set of pairs of $|S|$-vectors Λ with components between zero and one, we can specify a piecewise-linear convex function that is the solution to a POMDP update, without needing to identify Z, A, P, r, β, and Γ directly.

Let K be a set of $|S|$-vectors. Let V be a finite set of *variables*, and B be the set of all *bindings* mapping the elements of V to $\{1,2\}$. Let Λ be a set of $|S|$-vectors, $\Lambda = \{\lambda_k^z | z \in V, k \in \{1,2\}\}$. The sets K and Λ specify a piecewise-linear convex function over X by

$$f(x) = \max\{\max_{\sigma \in K}(x \cdot \sigma), \max_{b \in B}(x \cdot \alpha_b)\},$$

where we define $\alpha_b = \sum_{z \in V} \lambda_{b(z)}^z$ for $b \in B$.

The next theorem shows that any piecewise-linear convex function that can be specified this way can also be specified as the solution to a dynamic-programming update on some POMDP.

Theorem 9 *Given two sets of $|S|$-vectors ($|S| \geq 3$), K and $\Lambda = \{\lambda_k^z | z \in V, k \in \{1,2\}\}$ ($0 \leq \lambda_k^z[i] \leq 1, \forall i \in S$), there is a* POMDP *dynamic-programming-update problem such that*

$$\max_{\gamma \in \Gamma'}(x \cdot \gamma) = \max\{\max_{\sigma \in K}(x \cdot \sigma), \max_{b}(x \cdot \alpha_b)\}.$$

The POMDP *has $|Z| = |V|$, $|A| = |K| + 1$, and $|\Gamma| = 2$.*

Proof: The high-level idea is that there will be one action for each vector in K, with immediate rewards equal to the individual vectors in K. The α_b vectors will be formed by combinations of the vectors in Γ; a binding $b \in B$ is analogous to a table $\tau \in \mathcal{T}$.

We now define the POMDP in detail. The set of observations is the set of variables $Z = V$; there is one action for each vector in K and one corresponding to the Λ set, so $A = \{a_\sigma | \sigma \in K\} \cup \{a_0\}$. Let Γ be a set of two

$|S|$-vectors, γ_1 and γ_2, and let $\beta = 1$. Define $\gamma_i[i] = 2|V|$, for $i \in \{1,2\}$, and 0 otherwise.

For each $\sigma \in K$, the a_σ action results in an immediate reward of σ and a transition to system state 3 (which has zero value under Γ): for all $i \in S$, $r(a_\sigma)[i] = \sigma[i]$ and $P(z, a_\sigma)[i,j] = 1$ if $j = 3$ and zero otherwise. Since $\gamma_k[3] = 0$, for all $g \in \mathcal{T}$,

$$x \cdot [r(a_\sigma) + \beta \sum_z P(z, a_\sigma) \cdot g(z)] = x \cdot \sigma. \tag{2}$$

The a_0 action yields an immediate reward of zero and then causes a transition to system state 1, 2, or 3, under observation z, weighted according to the appropriate λ vector. Specifically, $r(a_0)[i] = 0$, for all $i \in S$ and $P(z, a_0)[i,j] = \lambda_j^z[i]/(2|V|)$ for $j \in \{1,2\}$, $P(z, a_0)[i,3] = (1/|V| - \lambda_1^z[i]/(2|V|) - \lambda_2^z[i]/(2|V|))$, and 0 otherwise. The P matrices are valid since each component is non-negative and for each i and a, $\sum_z \sum_j P(z,a)[i,j] = \sum_z 1/|V| = 1$.

The linear facets for action a_0 correspond to the α_b vectors in that

$$\begin{aligned}
&\max_{g \in \mathcal{T}} (x \cdot [r(a_0) + \beta \sum_z P(z, a_0) \cdot g(z)]) \\
&= \max_{b \in B} (x \cdot [\sum_z P(z, a_0) \cdot \gamma_{b(z)}]) \\
&= \max_{b \in B} (\sum_i (x[i] \sum_z \sum_j P(z, a_0)[i,j] \gamma_{b(z)}[j])) \\
&= \max_{b \in B} (\sum_i (x[i] \sum_z (P(z, a_0)[i,1]\gamma_{b(z)}[1] + P(z, a_0)[i,2]\gamma_{b(z)}[2] + P(z, a_0)[i,3]\gamma_{b(z)}[3]))) \\
&= \max_{b \in B} (\sum_i (x[i] \sum_z \lambda_{b(z)}^z[i])) \\
&= \max_{b \in B} (x \cdot \alpha_b) \ .
\end{aligned} \tag{3}$$

Combining Equations 1, 2, and 3, we have

$$\begin{aligned}
\max_{\gamma \in \Gamma'} (x \cdot \gamma) &= \max_{a \in A} \max_{g \in \mathcal{T}} (x \cdot [r(a) + \beta \sum_z P(z,a) \cdot g(z)]) \\
&= \max\{\max_{\sigma \in K} (x \cdot \sigma), \max_{b \in B} (x \cdot \alpha_b)\} \ ,
\end{aligned}$$

therefore the supplied vectors correspond exactly to the set of vectors obtained by performing a dynamic-programming on the given POMDP. □

A.2.1 Proof of Theorem 3

We want to show that there exists a family of POMDPs such that, for every n, $|S| = 2n$, $|A| = 1$, $|Z| = n$, $|\Gamma| = 2$, and $|\Gamma'| = 2^n$.

We use the specification described in Theorem 9. Let $S = \{1, \dots, 2n\}$, and $V = \{1, \dots, n\}$. For $z \in V$, and $k \in \{1, 2\}$, let $\lambda_k^z[2z + k - 2] = 1$, and 0 otherwise. Let $K = \emptyset$. By Theorem 9, there is a POMDP with $|A| = 1$, $|Z| = n$, and $|\Gamma| = 2$ such that $G = \bigcup_{b \in B} \{\sum_z \lambda_{b(z)}^z\}$. Note that $|G| = |B| = 2^{|V|} = 2^n$. We know that the set Γ' is a subset of G. We can show that, in fact, these sets are equal since every vector in G dominates all other vectors in G at at least one $x \in X$.

Lemma 3 *In the construction just described, for every $b \in B$ there is an $x \in X$ such that $x \cdot \alpha_b > x \cdot \alpha_{b'}$ for all $b' \neq b$.*

Proof: For a binding b, for each $z \in V$, let $x[2z + b(z) - 2] = 1/n$ and 0 otherwise. Note that $x \in X$. Now, for any $b' \in B$,

$$\begin{aligned}
x \cdot \alpha_{b'} &= x \cdot \sum_z \lambda_{b'(z)}^z \\
&= \sum_i (x[i] \sum_z \lambda_{b'(z)}^z[i]) \\
&= \sum_z \sum_i (x[i] \lambda_{b'(z)}^z[i]) \\
&= \sum_z x[2z + b'(z) - 2] \\
&= \sum_{z : b'(z) = b(z)} 1/n \quad ,
\end{aligned}$$

which is uniquely maximized for $b' = b$. □

Theorem 3 follows easily.

A.2.2 Example with an exponential number of vertices

We want to show that there is a family of POMDPs such that, for every n, $|S| = n + 1$, $|A| = 2n + 1$, $|Z| = 1$, $|\Gamma| = 1$, $|\Gamma'| \leq 2n + 1$, and the number of vertices in one of the linear regions of Hv is 2^n.

Once again, we will use the specification described in Theorem 9, inductively creating a separate piecewise-linear convex function for each n. For each n, we define a set K_n containing $2n + 1$ vectors, and we construct a vector σ^n such that the vectors in K_n form the walls of an n-dimensional hypercube bounding the region $R(\sigma^n, K_n)$. We define the family of POMDPs recursively, starting with $n = 1$. Let $S_1 = \{1, 2\}$, and $V = \{1\}$. Let $\lambda_k^z[i] = 0$ for all $z \in V$, $k \in \{1, 2\}$, and $i \in S$ (the set Λ is not needed in this construction). We use the notation $\langle x, y \rangle$ to concatenate two vectors (or a vector and scalar) into a single vector.

Let $K_1 = \{\langle 1, -1/2 \rangle, \langle 0, 0 \rangle, \langle -1, 0 \rangle\}$ and $\sigma^1 = \langle 0, 0 \rangle$. Let Ω_n be the set of extreme points of the region

$$\{x | x \cdot \sigma^n \geq x \cdot \sigma, \sigma \in K_n\}.$$

The extreme points are the information states where one of the other vectors in K_1 gives the same value as σ^1. It is easy to verify that

$$\Omega_1 = \{\langle 0, 1 \rangle, \langle 1/3, 2/3 \rangle\}.$$

Inductively define $\sigma^{n+1} = \langle \sigma^n, 0 \rangle$,

$$K_{n+1} = \{\langle \sigma, 0 \rangle : \sigma \in K_n\} \cup \{\langle 0, \ldots, 0, -1 \rangle, \langle 1, \ldots, 1, -1 \rangle\}$$

and $S_{n+1} = S_n \cup \{n + 2\}$.

Note that $\sigma^{n+1} = \langle 0, \ldots, 0 \rangle \in K_{n+1}$. The extreme points of σ^{n+1}'s region are the information states where $n + 1$ of the vectors in K_{n+1} give the same value as σ^{n+1}. Let $x \in \Omega_n$. Inductively, n vectors in K_n give the same value as σ^n at x. Let σ be any one of these vectors. It follows from the construction of σ that

$$\langle x, 0 \rangle \cdot \langle \sigma, 0 \rangle = x \cdot \sigma = 0 = x \cdot \sigma^n = \langle x, 0 \rangle \cdot \sigma^{n+1}$$

and

$$\langle x, 0 \rangle \cdot \langle 0, \ldots, 0, -1 \rangle = 0 = \langle x, 0 \rangle \cdot \sigma^{n+1},$$

so $\langle x, 0 \rangle \in X$ is an extreme point of σ^{n+1}'s region. Similarly,

$$\langle 1/2\ x, 1/2 \rangle \cdot \langle \sigma, 0 \rangle = 1/2\ x \cdot \sigma + 0 = 0 = \langle 1/2\ x, 1/2 \rangle \cdot \sigma^{n+1}$$

and

$$\langle 1/2\ x, 1/2 \rangle \cdot \langle 1, \ldots, 1, -1 \rangle = 1/2 - 1/2 = 0 = \langle 1/2\ x, 1/2 \rangle \cdot \sigma^{n+1},$$

so $\langle 1/2\ x, 1/2 \rangle \in X$ is an extreme point of σ^{n+1}'s region also. This means we can write

$$\Omega_{n+1} = \{\langle x, 0 \rangle : x \in \Omega_n\} \cup \{\langle 1/2\ x, 1/2 \rangle : x \in \Omega_n\}.$$

Since all the vectors in K_{n+1} and Ω_{n+1} are unique, $|K_n| = 2n + 1$ and $|\Omega_n| = 2^n$. Applying Theorem 9, there is a POMDP for each n with $|S| = n+1$, $|A| = 2n + 1$, $|Z| = 1$, $|\Gamma| = 1$, $|\Gamma'| = n + 1$, such that the number of vertices in one of the linear regions of Hv is 2^n.

A.3 Proof of Theorem 4

We want to show that the best algorithm for solving polynomially output-bounded POMDPs is efficient if only if $\mathcal{RP} = \mathcal{NP}$. To do this, we show a deep connection between this problem and the *unique-satisfying-assignment problem*, defined below.

A boolean formula in *conjunctive normal form* (CNF) is an "and" of a set of clauses where each clause is a set of "ors" of variables and negated variables. A *satisfying assignment* maps each of the variables to either "true" or "false" so the entire formula evaluates to "true." There is a corollary, proved by Valiant and Vazirani [14], that implies that there exists a polynomial-time algorithm for finding a satisfying assignment for a formula that is guaranteed to have at most one satisfying assignment only if $\mathcal{RP} = \mathcal{NP}$, a long-open problem.[3] We can show that a polynomial-time algorithm for solving polynomially output-bounded POMDPs could be used to solve the unique-satisfying-assignment problem in polynomial time (and vice versa), and therefore that such an algorithm exists if only if $\mathcal{RP} = \mathcal{NP}$.

We make use of the specification defined in Theorem 9 and define S, V, K, and Λ. Take a CNF formula consisting of a set of $M > 1$ clauses C, and variables V. The set of variables corresponds both to the variables in Theorem 9 and the boolean variables in the formula. Let $S = C \times V$. We write a system state as $(c, z) \in C \times V$. There is a pair of λ vectors for each variable $z \in V$, which will encode the formula. Vector λ_1^z indicates in which clauses variable z is unnegated and λ_2^z indicates in which clauses variable z is negated. More specifically, for each $z \in V$ and $k \in \{1, 2\}$, λ_k^z is a $|C \times V|$-vector with $\lambda_1^z[(c, z)] = 1$ if variable z appears unnegated in clause c and $\lambda_2^z[(c, z)] = 1$ if variable z appears negated in clause c. All other components of the λ vectors are zero.

For a binding $b \in B$, $b(z) = 1$ if variable z is true in the assignment and $b(z) = 2$ otherwise. Thus, if $\lambda_{b(z)}^z[(c, z)] = 1$, then clause c evaluates to "true" under binding b because of the binding of variable z in that clause.

[3]We thank Avrim Blum for pointing this out to us.

Let κ be a $|C \times V|$-vector with each component equal to $(M-1/2)/M$. For each $c \in C$, we define a $|C \times V|$-vector σ_c, as follows. For all $z \in V$, let $\sigma_c[(c,z)] = 1 + (M-1/2)/M - M$ and $\sigma_c[(c',z)] = 1 + (M-1/2)/M$ for $c' \neq c$. As described earlier, let $\alpha_b = \sum_z \lambda^z_{b(z)}$. From the definition of λ, every component of α_b is either a one or a zero.

Each α_b vector corresponds to an assignment and the σ_c and κ vectors are designed to jointly dominate all possible α_b vectors, except those correponding to satisfying assignments.

Lemma 4 *Let $K = \{\kappa\} \cup \{\sigma_c | c \in C\}$. There is an $x \in X$ such that $x \cdot \alpha_b > \max_{\sigma \in K}(x \cdot \sigma)$ if and only if b is a satisfying assignment.*

Proof: First, assume b is a satisfying assignment. We can construct an x^* such that $x^* \cdot \alpha_b = 1$, but $x^* \cdot \kappa < 1$ and $x^* \cdot \sigma_c < 1$ for all $c \in C$ as follows.

For each clause $c \in C$, pick a single variable z_c such that the binding of that variable in b causes clause c to be satisfied. Let $x^*[(c, z_c)] = 1/M$ for each $c \in C$ and 0 otherwise. Note that $x^* \in X$ because M components are set to $1/M$. Because of the zeros in x^* and the λ vectors,

$$\begin{aligned} x^* \cdot \alpha_b &= \sum_c \sum_z x^*[(c,z)] \sum_{z'} \lambda^{z'}_{b(z')}[(c,z)] \\ &= \sum_c \sum_z x^*[(c,z)] \lambda^z_{b(z)}[(c,z)] \\ &= \sum_c x^*[(c,z_c)] \lambda^{z_c}_{b(z_c)}[(c,z_c)] = \sum_c \frac{1}{M} = 1. \end{aligned}$$

Using the same x^* we see that $x^* \cdot \kappa = (M-1/2)/M < 1$, and for each clause $c \in C$,

$$\begin{aligned} x^* \cdot \sigma_c &= \sum_z \sum_{c'} x^*[(c',z)] \sigma_c[(c',z)] \\ &= \sum_z \left(x^*[(c,z)] \left(1 + \frac{M-\frac{1}{2}}{M} - M\right) + \sum_{c' \neq c} x^*[(c',z)] \left(1 + \frac{M-\frac{1}{2}}{M}\right)\right) \\ &= \sum_z \left(-M x^*[(c,z)] + \sum_{c'} x^*[(c',z)] \left(1 + \frac{M-\frac{1}{2}}{M}\right)\right) \\ &= -M\frac{1}{M} + \left(1 + \frac{M-\frac{1}{2}}{M}\right) \sum_z \sum_{c'} x^*[(c',z)] \\ &= -1 + 1 + \frac{M-\frac{1}{2}}{M} = \frac{M-\frac{1}{2}}{M} < 1. \end{aligned}$$

Thus, $x^* \cdot \alpha_b = 1 > \max_{\sigma \in K}(x^* \cdot \sigma)$, for satisfying assignment b.

Now we can show that, for a non-satisfying b, $x \cdot \alpha_b < \max_{\sigma \in K}(x \cdot \sigma)$ for all $x \in X$. We proceed by contradiction. Assume we have an $x \in X$ such that $\max_{\sigma \in K}(x \cdot \sigma) \leq x \cdot \alpha_b$ for a non-satisfying b. Then, $x \cdot \kappa \leq x \cdot \alpha_b$ and $x \cdot \sigma_c \leq x \cdot \alpha_b$ for all $c \in C$. Define $w_c = \sum_z x[(c, z)]$. We call w_c the *weight* of clause c, and note that $\sum_c w_c = 1$ if $x \in X$. Let $c^* = \arg\min_c w_c$; c^* is a minimum weight clause.

If $x \cdot \kappa \leq x \cdot \alpha_b$, then $(M - 1/2)/M \leq x \cdot \alpha_b = x \cdot [\sum_z \lambda^z_{b(z)}] \leq 1 - w_{c^*}$. The last inequality is justified by the fact that b is not satisfying so at least one clause contributes zero to the dot product and assuming it is the smallest weight clause gives us the largest possible value. This restricts w_{c^*} so that

$$w_{c^*} \leq \frac{1}{2M} < \frac{1}{2M} + \frac{1}{2(M-1)} = \frac{M - \frac{1}{2}}{M(M-1)}. \tag{4}$$

At the same time, it must be the case that $x \cdot \sigma_{c^*} \leq x \cdot \alpha_b$ which implies $1 + (M - 1/2)/M - M w_{c^*} \leq 1 - w_{c^*}$ and therefore $w_{c^*} \geq (M - 1/2)/(M(M - 1))$. This directly contradicts Inequality 4, and therefore we can conclude that, for a non-satisfying b, $x \cdot \alpha_b < \max_{\sigma \in K}(x \cdot \sigma)$ for all $x \in X$. □

By Theorem 9, there is a POMDP such that

$$\max_{\gamma \in \Gamma'}(x \cdot \gamma) = \max\left\{\max_{\sigma \in K}(x \cdot \sigma), \max_b(x \cdot \alpha_b)\right\}.$$

This POMDP is derived from the unique-satisfying-assignment-problem instance and has the property that $\Gamma' \not\subseteq K$ if and only if the boolean formula instance is satisfiable. Because we assumed the boolean formula has no more than 1 satisfying assignment, $|\Gamma'| \leq M + 2$; thus, the POMDP is polynomially output bounded.

Because the POMDP can be created in polynomial time, and the condition $\Gamma' \not\subseteq K$ can be checked in polynomial time, a polynomial-time algorithm for solving polynomially output-bounded POMDPs could be used to find unique satisfying assignments in polynomial time. As mentioned at the start of this section, this would imply $\mathcal{RP} = \mathcal{NP}$.

To complete the proof of Theorem 4, we need to argue that if $\mathcal{RP} = \mathcal{NP}$ then there is a randomized polynomial-time algorithm for solving polynomially output-bounded POMDPs. We can build up a set of vectors $U \subseteq \Gamma'$ one at a time by answering the question "Is there an information state x such that $\max_{\gamma \in U} x \cdot \gamma \neq [Hv](x)$?" and adding the dominating vector at

x into U. For polynomially output-bounded POMDPs, if the answer to this question is yes, then there is an x that can be written using polynomially many bits. Such an x can be identified in non-deterministic polynomial time using standard techniques, and therefore in randomized polynomial time if $\mathcal{RP} = \mathcal{NP}$. Because there are at most a polynomial number of vectors that can be added to U, the process terminates with $U = \Gamma'$ in polynomial time.

Acknowledgements

This work was supported in part by NSF grants IRI-9453383 and IRI-9312395 and Bellcore. The authors would like to thank John Hughes, Avrim Blum, and Philip Klein for their contributions to this work.

References

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

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

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

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

[5] William S. Lovejoy. A survey of algorithmic methods for partially observable Markov decision processes. *Annals of Operations Research*, 28:47–66, 1991.

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

[7] Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, August 1987.

[8] Ronald Parr and Stuart Russell. Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1995.

[9] Katsushige Sawaki and Akira Ichikawa. Optimal control for partially observable Markov decision processes over an infinite horizon. *Journal of the Operations Research Society of Japan*, 21(1):1–14, March 1978.

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

[11] 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.

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

[13] Edward J. Sondik. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research*, 26(2), 1978.

[14] L. G. Valiant and V. V. Vazirani. NP is as easy as detecting unique solutions. *Theoretical Computer Science*, 47(1):85–93, 1986.

[15] Chelsea C. White, III. Partially observed Markov decision processes: A survey. *Annals of Operations Research*, 32, 1991.

[16] Chelsea C. White, III and William T. Scherer. Solution procedures for partially observed Markov decision processes. *Operations Research*, 37(5):791–797, September-October 1989.