Approximate Planning for Factored POMDPs

Zhengzhu Feng
Computer Science Department
University of Massachusetts
Amherst MA 02062
fengzz@cs.umass.edu
New address: zhengzhu@gmail.com

Eric A. Hansen
Computer Science Department
Mississippi State University
Mississippi State MS 39762
hansen@cs.msstate.edu
New address: hansen@cse.msstate.edu

Abstract

We describe an approximate dynamic programming algorithm for partially observable Markov decision processes represented in factored form. Two complementary forms of approximation are used to simplify a piecewise linear and convex value function, where each linear facet of the function is represented compactly by an algebraic decision diagram. ln one form of approximation, the degree of state abstraction is increased by aggregating states with similar values. In the second form of approximation, the value function is simplified by removing linear facets that contribute marginally to value. We derive an error bound that applies to both forms of approximation. Experimental results show that this approach improves the performance of dynamic programming and extends the range of problems it can solve.

Introduction

Markov decision processes (MDPs) have been adopted as a framework for research in decision-theoretic planning (Boutilier *et al.* 1999). An MDP models planning problems for which actions have an uncertain effect on the state, and sensory feedback compensates for uncertainty about the state. An MDP is said to be completely observable if sensory feedback provides perfect state information before each action. It is said to be partially observable if sensory feedback is noisy and provides partial and imperfect state information. Although a partially observable Markov decision process (POMDP) provides a more realistic model, it is much more difficult to solve.

Dynamic programming is the most common approach to solving MDPs. However, a drawback of classic dynamic programming algorithms is that they require explicit enumeration of a problem's state space. Because the state space grows exponentially with the number of state variables, these algorithms are prey to Bellman's "curse of dimensionality." To address this problem, researchers have developed algorithms that exploit a factored state representation to create an abstract state space in which planning problems can be solved more efficiently. This approach was first developed for completely observable MDPs, using decision trees to aggregate states with identical values (Boutilier *et al.* 1995). Improved performance was subsequently achieved using algebraic decision diagrams (ADDs) in place of decision trees (Hoey *et al.* 1999). A factored representation has also been exploited in solving POMDPs using both decision trees (Boutilier & Poole 1996) and ADDs (Hansen & Feng 2000).

Although this approach improves the efficiency with which many problems can be solved, it provides no benefit unless there are states with identical values that can be aggregated. Moreover, the degree of state abstraction that can be achieved in this way may be insufficient to make a problem tractable. Many large MDPs and even small POMDPs resist exact solution. Thus, there is a need for approximation to allow a tradeoff between optimality and computation time. For completely observable MDPs, an approach to approximation that increases the degree of state abstraction by aggregating states with similar (rather than identical) values has been developed, using both decision trees (Boutilier & Dearden 1996) and ADDs (St-Aubin *et al.* 2000). This paper develops a related approach to approximation for the partially observed case.

Whereas the value function of a completely observable MDP can be represented by a single ADD, the value function of a POMDP is represented by a set of ADDs. This makes development of an approximation algorithm for factored POMDPs more complex. We describe two complimentary forms of approximation for POMDPs. The first is closely related to approximation in the completely observable case and involves simplifying an ADD by aggregating states of similar value. In the second form of approximation, the set of ADDs representing the value function is reduced in size, by removing ADDs that contribute only marginally to value. The latter technique is often used in practice, but has not been analyzed before in the literature. We clarify the relationship between these two forms of approximation, and derive a bound on their approximation error. Experimental results show that this approach to approximation can improve the rate of convergence of dynamic programming, as well as the range of problems it can solve.

Partially observable Markov decision processes

We assume a discrete-time POMDP with finite sets of states S, actions A, and observations O. The transition function $Pr(s'|s,a)$, observation function $Pr(o|s',a)$, and reward function $r(s,a)$ are defined in the usual way. We assume a discounted infinite-horizon optimality criterion with discount factor $\beta \in (0,1]$. A belief state, b, is a probability distribution over S maintained by Bayesian conditioning. As is well-known, it contains all information necessary for optimal action selection. This gives rise to the standard approach to solving POMDPs. The problem is recast as a completely observable MDP with a continuous, $|S|$-dimensional state space consisting of all possible belief states, denoted $\mathcal{B}$. In this form, the POMDP can be solved by iteration of a *dynamic programming operator* T that improves a value function $V : \mathcal{B} \rightarrow \mathcal{R}$ by performing the following "one-step backup" for all belief states b.

$$V_n(b) = TV_{n-1}(b) = \max_{a \in A}\{r(b,a) + \beta \sum_{o \in O} Pr(o|b,a)V_{n-1}(b_o^a)\}, \quad (1)$$

where $r(b,a) = \sum_{s \in S} b(s)r(s,a)$, b_o^a is the updated belief state after action a and observation o; and $Pr(o|b,a) = \sum_{s,s' \in S} Pr(o,s'|s,a)b(s)$ where $Pr(o,s'|s,a) = Pr(s'|s,a)Pr(o|s',a)$.

The theory of dynamic programming tells us that the optimal value function V^* is the unique solution of the equation system $V = TV$, and that $V^* = \lim_{n \rightarrow \infty} T_n V_0$, where T_n denotes n applications of operator T to any initial value function V_0. We note that T_n corresponds to the value iteration algorithm.

An important result of Smallwood and Sondik (Smallwood & Sondik 1973) is that the dynamic-programming operator T preserves the piecewise linearity and convexity of the value function. A piecewise linear and convex value function V can be represented by a finite set of $|S|$-dimensional vectors of real numbers, $\mathcal{V} = \{v^0, v^1, \ldots, v^k\}$, such that the value of each belief state b is defined as:

$$V(b) = \max_{0 \leq i \leq k} \sum_{s \in S} b(s)v^i(s).$$

Several algorithms for computing the dynamic-programming operator have been developed. The most efficient, called incremental pruning (Cassandra *et al.* 1997), relies on the fact that the updated value function V_n of Equation (1) can be defined as a combination of simpler value functions, as follows:

$$V_n(b) = \max_{a \in A} V_n^a(b)$$

$$V_n^a(b) = \sum_{o \in O} V_n^{a,o}(b)$$

$$V_n^{a,o}(b) = \frac{r(b,a)}{|O|} + \beta Pr(o|b,a)V_{n-1}(b_o^a)$$

Each of these value functions is also piecewise linear and convex, and can be represented by a unique minimum-size set of vectors, denoted $\mathcal{V}_n$, $\mathcal{V}_n^a$, and $\mathcal{V}_n^{a,o}$ respectively.

The *cross sum* of two sets of vectors, $\mathcal{U}$ and $\mathcal{W}$, is defined as $\mathcal{U} \oplus \mathcal{W} = \{u + w | u \in \mathcal{U}, w \in \mathcal{W}\}$. An operator that takes a set of vectors $\mathcal{U}$ and reduces it to its unique minimum form is denoted PRUNE($\mathcal{U}$). Using this notation, the minimum-size sets of vectors defined above can be computed as follows:

$$V_n = PRUNE(\cup_{a \in A} \mathcal{V}_n^a)$$
$$\mathcal{V}_n^a = PRUNE(\oplus_{o \in O} \mathcal{V}_n^{a,o})$$
$$\mathcal{V}_n^{a,o} = PRUNE(\{v^{a,o,i} | v^i \in \mathcal{V}_{n-1}\}),$$

where $v^{a,o,i}$ is the vector defined by

$$v^{a,o,i}(s) = \frac{r(s,a)}{|O|} + \beta \sum_{s' \in S} Pr(o,s'|s,a)v^i(s'). \quad (2)$$

Incremental pruning gains its efficiency (and its name) from the way it interleaves pruning and cross-sum to compute V_n^a, as follows:

$$V_n^a = PRUNE(\ldots PRUNE(PRUNE(\mathcal{V}_n^{a,o1} \oplus \mathcal{V}_n^{a,o2}) \oplus \mathcal{V}_n^{a,o3}) \ldots \mathcal{V}_n^{a,o_k}).$$

PRUNE reduces a set of vectors to a unique, minimal-size set by removing "dominated" vectors, that is, vectors that can be removed without affecting the value of any belief state. There are two tests for dominated vectors. One test determines that vector w is dominated by some other vector $u \in \mathcal{V}$ when

$$w(s) \leq u(s), \ \forall s \in S. \quad (3)$$

Although this test can be performed efficiently, it cannot detect all dominated vectors. Therefore, it is supplemented by a second, less efficient test that determines that a vector w is dominated by a set of vectors $\mathcal{V}$ when the following linear program cannot be solved for a value of d that is greater than zero.

variables: $d, b(s) \forall s \in S$
maximize d
subject to the constraints
$\sum_{s \in S} b(s)(w(s) - u(s)) \geq d, \ \forall u \in \mathcal{V}$
$\sum_{s \in S} b(s) = 1$

In a single iteration of incremental pruning, many linear programs must be solved to detect all dominated vectors, and the use of linear programming to test for domination has been found to consume more than 95% of the running time of incremental pruning (Cassandra *et al.* 1997). The number of variables in each linear program is determined by the size of the state space. The number of constraints in each linear program (as well as the number of linear programs that need to be solved) is determined by the size of the vector set being pruned. This reflects the two principal sources of complexity in solving POMDPs. One source of complexity is shared by completely observable MDPs: the size of the state space.

The other source of complexity is unique to POMDPs: the number of linear functions needed to represent the piecewise linear and convex value function.

In this paper, we explore two forms of approximation that address these two sources of complexity. We use state abstraction to reduce the number of variables in each linear program. We use relaxed tests for dominance to reduce the number of constraints in each linear program, as well as the number of linear programs that need to be solved. Before describing our approximation algorithm, we describe the framework for state abstraction.

State abstraction and algebraic decision diagrams

We consider an approach to state abstraction for MDPs and POMDPs that exploits a factored representation of the problem. We assume the relevant properties of a domain are described by a finite set of Boolean state variables, $\mathcal{X} = \{X_1, \ldots, X_n\}$, and observations are described by a finite set of Boolean observation variables $\mathcal{Y} = \{Y_1, \ldots, Y_m\}$. We define an abstract state as a partial assignment of truth values to $\mathcal{X}$, corresponding to a set of possible states.

Algebraic decision diagrams Instead of using matrices and vectors to represent the POMDP model, we use a data structure called an *algebraic decision diagram* (ADD) that exploits state abstraction to represent the model more compactly. Decision diagrams are widely used in VLSI CAD to represent and evaluate large state space systems (Bahar *et al.* 1993). A binary decision diagram is a compact representation of a Boolean function, $\mathcal{B}^n \rightarrow \mathcal{B}$. An algebraic decision diagram (ADD) generalizes a binary decision diagram to represent real-valued functions, $\mathcal{B}^n \rightarrow \mathcal{R}$. Operations such as sum, product, and expectation (corresponding to similar operations on matrices and vectors) can be performed on ADDs, and efficient packages for manipulating ADDs are available (Somenzi 1998). Hoey et al, (Hoey *et al.* 1999) show how to use ADDs to represent and solve completely observable MDPs. Hansen and Feng (Hansen & Feng 2000) extend this approach to POMDPs, based on earlier work of Boutilier and Poole (Boutilier & Poole 1996). In the rest of this section, we summarize this approach to state abstraction for POMDPs.

Model We represent the state transition function for each action a using a two-slice dynamic belief network (DBN). The DBN has two sets of variables, one set $\mathcal{X} = \{X_1, \ldots, X_n\}$ refers to the state before taking action a, and the other set $\mathcal{X}' = \{X'_1, \ldots, X'_n\}$ refers to the state after.

For each post-action variable X'_i, the conditional probability function $P^a(X'_i|\mathcal{X})$ of the DBN is represented compactly using an ADD. It is convenient to construct a single ADD, $P^a(\mathcal{X}'|\mathcal{X})$, that represents in factored form the state transition function for all post-action variables. Hoey et al. (Hoey *et al.* 1999) call this a *complete action diagram* and describe the steps required to construct it.

The observation model of a POMDP is represented in factored form, in a similar way. We use an ADD $P^a(Y_i|\mathcal{X}')$ to represent the probability that observation variable Y_i is true after action a is taken and the state variables change to $\mathcal{X}'$. Given an ADD, $P^a(Y_i|\mathcal{X}')$, for each observation variable Y_i, it is again convenient to construct a single ADD, $P^a(\mathcal{Y}|\mathcal{X}')$, that represents in factored form the observation function for all observation variables. We call this a *complete observation diagram*, and it is constructed in the same way as a complete action diagram.

Given a complete action diagram and a complete observation diagram, a single ADD, $P^{a,o}(\mathcal{X}'|\mathcal{X})$, representing the transition probabilities for all state variables after action a and observation o, is constructed as follows:

$$P^{a,o}(\mathcal{X}'|\mathcal{X}) = P^a(\mathcal{X}'|\mathcal{X})P^a(\mathcal{Y}|\mathcal{X}').$$

The ADD $P^{a,o}(\mathcal{X}'|\mathcal{X})$ represents the probabilities $Pr(o, s'|s, a)$, just as $P^a(\mathcal{X}'|\mathcal{X})$ represents the probabilities $Pr(s'|s, a)$ and $P^a(\mathcal{Y}|\mathcal{X}')$ represents the probabilities $Pr(o|s', a)$.

The reward function for each action a can also be represented compactly by an ADD, denoted $R^a(\mathcal{X})$. Similarly, a piecewise linear and convex value function for a POMDP can be repesented compactly by a set of ADDs. We use the notation $\mathcal{V} = \{v^1(\mathcal{X}), v^2(\mathcal{X}), \ldots, v^k(\mathcal{X})\}$ to denote this value function.

Dynamic programming Hansen and Feng (Hansen & Feng 2000) describe how to modify the incremental pruning algorithm to exploit this factored representation of a POMDP for computational speedup. We briefly review their approach, and refer to their paper for details.

The first step of incremental pruning is generation of the linear functions in the sets $\mathcal{V}_n^{a,o}$. For a POMDP represented in factored form, the following equation replaces Equation (2):

$$v^{a,o,i}(\mathcal{X}) = \frac{R^a(\mathcal{X})}{|O|} + \beta \sum_{\mathcal{X}'} P^{a,o}(\mathcal{X}'|\mathcal{X})v^i(\mathcal{X}').$$

All terms in this equation are represented by ADDs. The symbol $\sum_{\mathcal{X}'}$, denotes an ADD operator called existential abstraction that sums over the values of the state variables in $P^{a,o}(\mathcal{X}'|\mathcal{X})v^i(\mathcal{X}')$, exploiting state abstraction to compute the expected value efficiently.

State abstraction is also exploited to perform pruning more efficiently. Recall that the value function is represented by a set of linear functions, Each linear function is represented compactly by an ADD that can map multiple states to the same value, corresponding to a leaf of the ADD. In this case, the leaf corresponds to an abstract state. Hansen and Feng (Hansen & Feng 2000) describe an algorithm that finds a partition of the state space into abstract states that is consistent with the set of ADDs. Given this abstract state space, both tests for dominance can be performed more efficiently. In particular, the number of variables in the linear program used to test for dominace is reduced in proportion to the reduction in size of the state space. Because linear programming consumes most of the running time of incremental pruning, this significantly improves the performance of the algorithm in the best case. In the worst case, performance is only slightly worse since the overhead for this approach is almost negligible. Hansen and Feng (Hansen & Feng 2000)

report speedups of up to a factor of twenty for the problems they test. The degree of speedup is proportional to the degree of state abstraction.

Approximation algorithm

As an approach to scaling up dynamic programming for POMDPs, the exact algorithm reviewed in the previous section has two limitations. First, there may not be sufficiently many (or even any) states with identical values to create an abstract state space that is small enough to be tractable. Second, although the size of the state space contributes to the complexity of POMDPs, the primary source of complexity is the potential exponential growth in the number of linear functions (ADDs) needed to represent the value function.

We now describe an approximate dynamic programming algorithm that addresses both of these limitations by ignoring differences of value less than some error threshold 6. lt runs more efficiently than the exact algorithm because it computes a simpler, approximate value function for which we can bound the approximation error. There are two places in which the algorithm ignores value differences – in representing state values, and in representing values of belief states. These correspond to two complementary forms of approximation, one in which an ADD representing state values is simplified, and the other in which a set of ADDs representing belief state values is reduced in size. In other words, one form of approximation reduces the size of the state space (using state abstraction) and the other reduces the size of the value function (by pruning more aggressively). Before describing the algorithm, we define what we mean by approximate dynamic programming and present some theoretical results that allow us to bound the approximation error.

Approximation with bounded error We begin by defining what we mean by an approximate value function and an approximate dynamic programming operator.

Definition 1. *A value function $\hat{V}$ approximates a value function V with approximation error δ if $||V - \hat{V}|| \leq \delta$. (Note that $||V - \hat{V}||$ denotes $\max_{b \in \mathcal{B}} |V(b) - \hat{V}(b)|$).*

Definition 2. *An operator $\hat{T}$ approximates the dynamic programming operator T if for any value function V, $||TV - \hat{T}V|| \leq \delta$.*

We define an approximate value iteration algorithm $\hat{T}_n$ in the same way that we defined the value iteration algorithm T_n. The error between the approximate and exact n-step value functions is bounded as follows.

Theorem 1. *For any $n > 0$ and any value function V,* $||T_n V - \hat{T}V|| \leq \frac{\delta}{1-\beta}$

To compute a bound on the error between a n-step approximate value function and the optimal value function, we use the Bellman residual between the $(n-1)$th and nth approximate value functions. The following theorem is essentially the same as Theorem 12.2.5 of Ortega and Rheinboldt (Ortega & Rheinboldt 1970), who studied approximate contraction mappings for systems of nonlinear equations, and Theorem 4.2 of Cheng (Cheng 1998), who first applied their result to POMDPs.

Theorem 2. *The error between the current and optimal value function is bounded as follows,*

$$||\hat{T}_n V - V^*|| \leq \frac{\beta}{1-\beta} ||\hat{T}_n V - \hat{T}_{n-1} V|| + \frac{\delta}{1-\beta}.$$

Value iteration using an approximate dynamic programming operator converges "weakly," that is, two successive value functions fall within a distance $\frac{2\delta}{1-\beta}$ in the limit. (Decreasing δ after "weak" convergence will allow further improvement, as discussed later.)

Theorem 3. *For any value function V and $\epsilon > 0$, there is an N such that for all $n > N$,*

$$||\hat{T}_n V - \hat{T}_{n-1} V|| \leq \frac{2\delta}{1-\beta} + \epsilon$$

Simplifying ADDs We first describe an approach to approximation that simplifies an ADD by ignoring small differences in state values. It is based on a similar approach to approximation for completely observable MDPs (St-Aubin *et al.* 2000), although modifications are needed to extend this approach to POMDPs.

For completely observable MDPs, a single ADD represents the value function. Each leaf of the ADD corresponds to a distinct value. If more than one state has the same value, the states are mapped to the same leaf. In this way, a leaf can represent a set of states, or equivalently, an abstract state. Because state abstraction can be exploited to accelerate dynamic programming, the approach to approximation is to increase the degree of state abstraction by aggregating states with similar (though not identical) values.

St-Aubin et al. (St-Aubin *et al.* 2000) introduce the following notation and terminology. The value of a state is represented as a pair $[l, u]$, where the lower, l, and upper, u, bounds on the values are both represented. The *span* of a state, s, is given by $span(s) = u - l$. The *combined span* of states $s_1, s_2, \ldots, s_n$ with values $[l_1, u_1], \ldots, [l_n, u_n]$, is given by $cspan(s_1, s_2, \ldots, s_n) = \max(u_1, \ldots, u_n) - \min(l_1, \ldots, l_n)$. The method of approximation is to merge states (and correspondingly, leaves of an ADD) when their combined span is less than δ. This approximation is performed after each iteration of dynamic programming, The simpler ADD allows computational speedup, at the cost of some approximation error introduced by ignoring differences of value less than δ.

To implement this approach to approximation, St-Aubin et al. (St-Aubin *et al.* 2000) modified the ADD package so that a leaf of an ADD can represent a range of values, and a single ADD can represent a *ranged value function*. We don't do this because the value function of a POMDP is represented by a set of ADDs, instead of a single ADD, and we are concerned with upper and lower bounds on the values of belief states. The set of ADDs representing the lower bound function may not be the same as the set of ADDs representing the upper bound function. An alternative to a ranged value function is to use two ADDs to represent bounds on state values – one for lower bounds and one for upper bounds. But this representation would require

performing incremental pruning twice – once to compute a piecewise linear and convex lower bound function and once to compute a piecewise linear and convex upper bound function – doubling the complexity of the algorithm. Instead, we found that we can achieve an equally good result by computing a piecewise linear and convex lower bound function only, and representing the upper bound by using a scalar for the approximation error.

Figure 1 illustrates the effect of simplification. The ADD on the left is simplified by merging leaves that have a combined span of less than $\delta = 0.5$. The ADD on the right represents a lower bound on the value of each abstract state. Adding δ to each lower bound gives the upper bound. We use the following algorithm to simplify an ADD. The input is an ADD and approximation threshold δ. The output is a simplified ADD with bounded error.

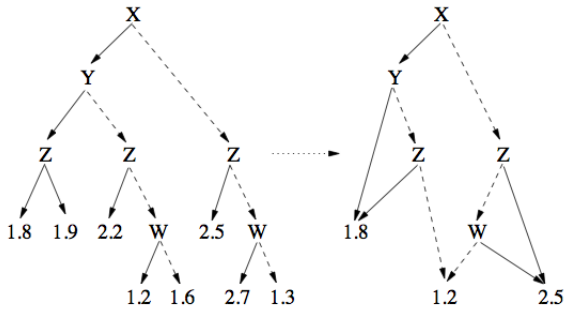

Figure 1: Example of ADD simplification.

QUEUE ←all leaves of ADD sorted in increasing order of value
$s \leftarrow$ remove first element from QUEUE
$X \leftarrow \{s\}$
While QUEUE is not empty
  $t \leftarrow$ remove next element from QUEUE
  if cspan($X \cup \{t\}$) $\leq \delta$
    $X \leftarrow X \cup \{t\}$
  else
    merge(X) and create new ADD leaf for X
    $X \leftarrow \{t\}$
  endif
endwhile

Because the complexity of each merge is $|S|$, the complexity of this algorithm is $O(|S|^2)$. Performing this simplification algorithm on each ADD in a piecewise linear and convex value function results in an approximate value function with approximation error δ.

Theorem 4. *Let $V = \{v_1, \ldots, v_n\}$ be a piecewise linear and convex value function and let $V' = \{v'_1, \ldots, v'_n\}$ be its approximation such that for each v'_i, we have $||v_i - v'_i|| \leq \delta$. Then $||V - V'|| \leq \delta$.*

Pruning ADDs In the second Section, we described two tests for dominated linear functions. It has long been recognized that both tests are sensitive to numerical imprecision errors when the value representing the degree of dominace is close to zero. Thus, a precision parameter is typically used to prevent linear functions from being included in the value function due to numerical imprecision error, Instead of testing for a value greater than zero to ensure that a linear function is not dominated, the test is for a value greater that 10^{-15}, for example, or some number that represent the limit of numerical precision on the computer.

Many practitioners have noticed that increasing this precision parameter (to a value of, say, 10^{-5}), has the added benefit of pruning vectors that contribute only marginally to the value function, This often results in significant performance improvement. Although this technique is widely used in practice, the effect of this approximation on the error bound of the value function has not been analyzed before in the literature. We consider this second method of approximation in this paper because of its close, and complementary, relationship to our first method of approximation, which also ignores small differences of value. Equation 3 gives a test for dominance that we generalize to allow approximation as follows.

Definition 3. *A linear function w is approximately dominated by another linear function $u \in \mathcal{V}$ when*

$$w(s) - \delta \leq u(s), \ \forall s \in S,$$

where $\delta > 0$.

The linear programming test for domination is generalized to allow approximation as follows.

Definition 4. *A linear function w is approximately dominated by a set of linear functions $\mathcal{V}$, if the output of the linear program, d, is less than $\delta > 0$.*

Let $PRUNE'$ be the pruning operator that employs these two approximate dominance tests.

Theorem 5. *For any set of vectors $\mathcal{V}$,*

$$||PRUNE(\mathcal{V}) - PRUNE'(\mathcal{V})|| \leq \delta.$$

Accumulation of error Both the approximate ADD simplification algorithm and the approximate pruning algorithm are applied repeatedly during incremental pruning. They are applied to each set $\mathcal{V}_n^{a,o}$. They are applied to each of the $|O|$ sets of ADDs created by the cross-sum operator during the computation of $\mathcal{V}_n^a$. Finally, they are applied to the set $\mathcal{V}_n$ created by the union of the sets $\mathcal{V}_n^a$. Thus, we must consider how approximation error accumulates during the progress of incremental pruning.

Lemma 1. *If a set of ADDs representing value function V is simplified with approximation error δ^1 and then pruned with approximation error δ^2, the resulting set of ADDs represents a value function that approximates V with error $\delta^1 - \delta^2$.*

Lemma 2. *If $\hat{V}^1$ is an approximation of value function V^1 with approximation error δ^1, and $\hat{V}^2$ is an approximation of value function V^2 with approximation error δ^2, then:*

1. *$\hat{V}^1 + \hat{V}^2$ is an approximation of $V^1 + V^2$ with approximation error $\delta^1 + \delta^2$, and*

2. $\hat{V}^1 \cup \hat{V}^2$ *is an approximation of* $V^1 \cup V^2$ *with approximation error* $\max(\delta^1, \delta^2)$.

Theorem 6. *Let* $\hat{T}$ *denote an approximation of the dynamic programming operator* T *computed by incremental pruning with simplification error* δ^1 *and pruning error* δ^2*. For any value function* V*,*

$$||TV - \hat{T}V|| \leq (2|O| + 1)(\delta^1 + \delta^2).$$

Letting $\delta = (2|O|+1)(\delta^1+\delta^2)$, we can use Theorem 2 to compute a bound on the error between the approximate and optimal value functions.

Adjustment of approximation Finally, we note that with exact dynamic programming, the difference between successive value functions always decreases from one iteration to the next. This is not necessarily the case with approximate dynamic programming. It suggests a strategy for reducing the approximation parameters over successive iterations. Whenever there is an increase in the Bellman residual, we reduce the approximation parameters (e.g., by the discount factor 0.5) and the solution continues to improve. By using a high degree of approximation initially and gradually reducing it, we may accelerate the rate of improvement in initial iterations and still eventually achieve a result of equal quality as a result found by the exact algorithm. This is explored in the next section.

Analysis of performance

Hansen and Feng (Hansen & Feng 2000) use seven test problems to evaluate the performance of their exact dynamic programming algorithm for factored POMDPs. Among these, the fourth test problem (with six state variables, five actions, and two observation variables) illustrates the Worstcase performance of the algorithm. Because there are strong dependencies among all the state variables, the algorithm finds no state abstraction. So we use this example as a test of whether the approximation algorithm can create state abstractions where the exact algorithm cannot. Figure 2 compares the size of the abstract state space created by the exact and approximation algorithms over successive iterations. Approximate ADD simplification makes it possible to solve the problem in an abstract state space that varies in size from five to thirty states, compared to 64 states in the original state space. This shows that ADD simplification can create useful state abstractions for problems with little or no variable independence.

Figure 3 and 4 show the interaction between the two forms of approximation by way of their effect on the two principal sources of POMDP complexity – the size of the state space and the size of the value function. The data is collected by running the program with different pruning and simplification errors for 30 iterations. The average size of the abstract state space and the average size of the value function are then plotted as a function of the two types of approximation error. Figure 3 shows that increasing the ADD pruning error has little or no effect on ADD simplification. Figure 4 shows that increasing the ADD simplification error only slightly amplifies the effect of ADD pruning. Thus, the performance improvement achieved from each form of approximation is almost independent, with a slight positive interaction effect. The two methods of approximation are complementary. ADD simplification decreases the size of the state space and ADD pruning decreases the size of the value function.

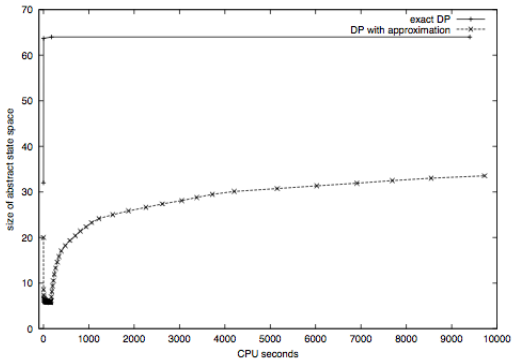

Figure 2: Size of abstract state space using exact and approximate ADD simplification.

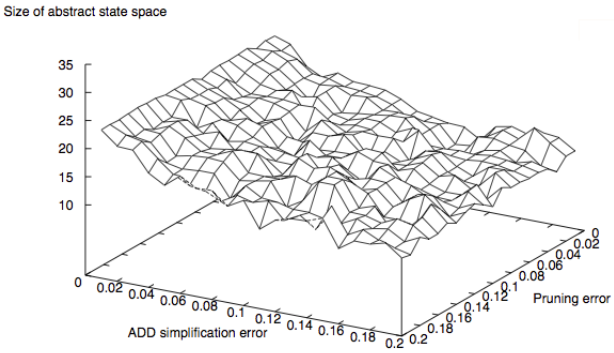

Figure 3: Interaction between ADD simplification and pruning error on size of abstract state space.

Figure 5 shows the separate effect of each form of approximation on the rate of convergence, as well as their combined effect. (The ADD simplification error is 0.1 and the ADD pruning error is 0.0l.) lt shows that using both forms of approximation results in better performance than using either one alone. It also shows that the approximation algorithm can find a better solution than the exact algorithm, in the same amount of time. The reason for this is that the approximation algorithm approximates the dynamic-programming operator, which performs a single iteration of dynamic programming. Dynamic programming takes many iterations to converge. Because approximation allows the dynamic-programming operator to be computed faster in exchange for slightly less improvement of the value function, approximation can have the effect of increasing the *rate* of improvement. In other words, the approximation algorithm can perform more iterations in the same amount of time, and, as a result, can find a better solution in the same amount of time. This is true even though each iteration of the approximation

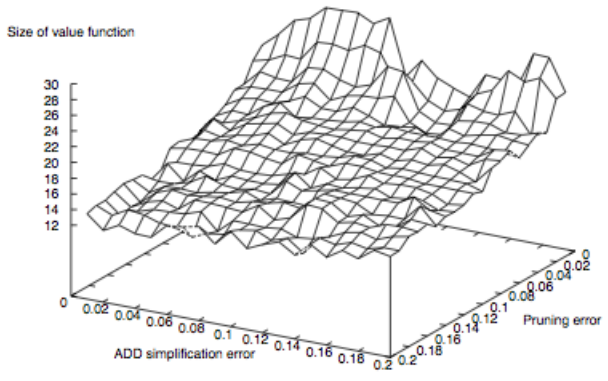

Figure 4: Interaction between ADD simplification and pruning error on size of value function.

algorithm may not improve the value function as much as a corresponding iteration of the exact algorithm.

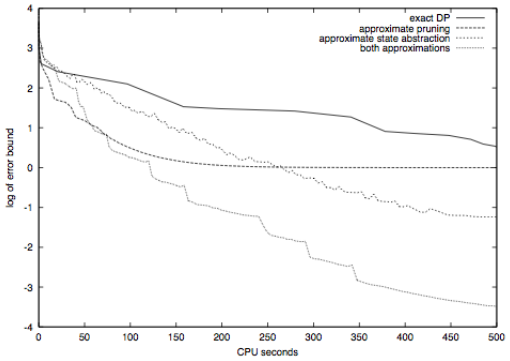

Figure 5: Rate of convergence using both forms of approximation, separately and together..

We are not only interested in improving the rate of convergence of dynamic programming. We are also interested in solving larger problems than the exact algorithm can solve. The scalability of both the exact algorithm and the approximation algorithm is limited by the same two factors – the size of the state space and the size of the value function. (The dynamic programming algorithm currently cannot handle problems with more than about 50 states or value functions with more than a few hundred ADDs.) The approximation algorithm scales better than the exact algorithm because it can control the size of the state space and the size of the value function. It controls the size of the (abstract) state space by using approximation to adjust the degree of state abstraction. It controls the size of the value function by using approximation to adjust the threshold for pruning ADDs, and thus the number of ADDs that are pruned. This allows the algorithm to find approximate solutions to problems with more states than the exact algorithm can handle, and to avoid an exponential explosion in the size of the value function. The quality of the solution that can be found within these limitations is problem-dependent, but easily estimated by computing the error bound.

Conclusion

POMDPs are very difficult to solve exactly and it is widely-recognized that approximation is needed to solve realistic problems. We have described two complementary forms of approximation that improve the performance of a dynamic programming algorithm that computes a piecewise linear and convex value function. The first form of approximation increases the degree of state abstraction by ignoring state distinctions that have little effect on value. The second form of approximation reduces the number of linear functions used to represent the value function by removing those that have little effect on value. Both forms of approximation allow computational speedup in exchange for a bounded decrease in solution quality. Both also have tunable parameters that allow the degree of approximation to be adjusted to suit the problem. We showed that this approach to approximation improves both the rate of convergence of dynamic programming; and its scalability.

References

Bahar; R.I.; Frohm; E.A.; Gaona; C.M.; Hachtel; G.D.; Macii; E.; Pardo; A.; and Somenzi; F. 1993. Algebraic decision diagrams and their applications. In *Proceedings of the International Conference on ComputerAided Design*. pp 188-191.

Boutilier; C.; Dean; T.; and Hanks; S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. In *Journal of Artificial Intelligence Research*, 11:1-94.

Boutilier; C.; Dearden; R.; and Goldszmidt; M. 1995. Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence (IJCAI-95)*. pp 1104-1111.

Boutilier; C. and Poole; D. 1996. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*. pp 1168-1175.

Boutilier; C. and Dearden; R. 1996. Approximating value trees in structured dynamic programming. In *Proceedings of the Fourteenth International Conference on Machine Learning*. pp 54-62.

Cassandra; A.R.; Littman; M.L.; and Zhang; N.L. 1997. Incremental pruning: A simple; fast; exact method for partially observable Markov decision processes. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-97)*. pp 54-61.

Cheng; H. 1998. Algorithms for Partially Observable Markov Decision Processes. PhD Thesis, University of British Columbia.

Hansen; E. and Feng; Z. 2000. Dynamic programming for POMDPs using a factored state representation. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pp 130-139.

Hoey; J.; St-Aubin; R.; Hu; A.; and Boutilier; C. 1999. SPUDD: Stochastic Planning using Decision Diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI-99)*.

Ortega; J.M and Rheinboldt; W.C. 1970. Iterative Solution of Nonlinear Equations in Several Variables. Academic Press, New York.

Smallwood; R.D. and Sondik; E.J. 1973. The optimal control of partially observable Markov processes over a finite horizon. In *Operations Research* 21:1071-1088.

Somenzi; F. 1998. CUDD: CU decision diagram package. Available from ftp://vlsi.colorado.edu/pub/.

St-Aubin; R.; Hoey; J.; and Boutilier; C. 2000. APRICODD: Approximate Policy Construction using Decision Diagrams. In *Advances in Neural Information Processing Systems 13 (NIPS-00): Proceedings of the 2000 Conference*.