An Approach to State Aggregation for POMDPs

Zhengzhu Feng
Computer Science Department
University of Massachusetts
Amherst, MA 01003
fengzz@cs.umass.edu

Eric A. Hansen
Dept. of Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
hansen@cse.msstate.edu

Abstract

A partially observable Markov decision process (POMDP) provides an elegant model for problems of planning under uncertainty. Solving POMDPs is very computationally challenging, however, and improving the scalability of POMDP algorithms is an important research problem. One way to reduce the computational complexity of planning using POMDPs is by using state aggregation to reduce the (effective) size of the state space. State aggregation techniques that rely on a factored representation of a POMDP have been developed in previous work. In this paper, we describe similar techniques that do not rely on a factored representation. These techniques are simpler to implement and make this approach to reducing the complexity of POMDPs more general. We describe state aggregation techniques that allow both exact and approximate solution of non-factored POMDPs and demonstrate their effectiveness on a range of benchmark problems.

Introduction

The problem of planning in domains in which actions have uncertain effects and sensors provide imperfect state information can be formalized as the problem of solving a partially observable Markov decision process (POMDP). Dynamic programming (DP) is a standard approach to solving POMDPs. However, it is very computationally-intensive and limited to solving small problems exactly. Because the complexity of DP depends in part on the size of the state space, one way to improve its efficiency is by using state aggregation to reduce the (effective) size of the state space.

In previous work, state aggregation techniques have been developed that significantly improve the efficiency of DP for POMDPs (Boutilier & Poole 1996; Hansen & Feng 2000; Guestrin, Koller, & Parr 2001; Feng & Hansen 2001). However, they rely on a factored representation of the POMDP in which the state space is modeled by a set of state variables. (Often, the state variables are assumed to be Boolean, although this is not necessary.) In this paper, we show that most (though not all) of the benefit of this approach can be achieved by a much simpler and more direct approach to state aggregation that does not assume a factored representation.

Background

We consider a discrete-time POMDP with a finite set of states, S, a finite set of actions, A, and a finite set of observations, $\mathcal{O}$. Each time period, the environment is in some state $s \in S$, the agent takes an action $a \in A$, the environment makes a transition to state $s' \in S$ with probability $Pr(s'|s,a) \in [0,1]$, and the agent observes $o \in \mathcal{O}$ with probability $Pr(o|s',a) \in [0,1]$. In addition, the agent receives an immediate reward with expected value $r(s,a) \in \Re$. We assume the objective is to maximize expected total discounted reward over an infinite horizon, where $\beta \in [0,1)$ is the discount factor.

Although the state of the environment cannot be directly observed, the probability that it is in a given state can be calculated. Let b denote a vector of state probabilities, called a *belief state*, where $b(s)$ denotes the probability that the system is in state s. If action a is taken and followed by observation o, the successor belief state, denoted b_o^a, is determined by revising each state probability using Bayes' theorem, as follows,

$$b_o^a(s') = \frac{\sum_{s \in S} Pr(s', o|s, a) b(s)}{\sum_{s,s' \in S} Pr(s', o|s, a) b(s)} \quad (1)$$

where $Pr(s', o|s, a) = Pr(s'|s, a) Pr(o|s', a)$. From now on, we adopt the simplified notation, $Pr(o|b, a) = \sum_{s,s' \in S} Pr(s', o|s, a) b(s)$, to refer to the normalizing factor in the denominator. It is well-known that a belief state updated by Bayesian conditioning is a sufficient statistic that summarizes all information necessary for optimal action selection. This gives rise to the standard approach to solving POMDPs. The problem is transformed into an equivalent, completely observable MDP with a continuous, $|S|$-dimensional state space consisting of all possible belief states, denoted $\mathcal{B}$. In this form, a POMDP can be solved by iteration of a *dynamic programming operator* T that improves a value function $V : \mathcal{B} \rightarrow \Re$ by performing the following "one-step backup" for all belief states $b \in \mathcal{B}$.

$$V'(b) = T(V) = \max_{a \in A} \left\{ r(b,a) + \beta \sum_{o \in \mathcal{O}} Pr(o|b,a) V(b_o^a) \right\}, \quad (2)$$

where $r(b,a) = \sum_s b(s) r(s,a)$ is the immediate reward for taking action a in belief state b.

The dynamic-programming operator is the core step of *value iteration*, a standard algorithm for solving infinite-horizon POMDPs. Value iteration solves a POMDP by repeatedly applying the dynamic-programming operator to improve the value function. By the theory of dynamic programming, the optimal value function V^* is the unique solution of the equation system $V = TV$, and $V^* = \lim_{n\to\infty} T_n V_0$, where T_n denotes n applications of operator T to any initial value function V_0. A value function V is said to be ϵ-optimal if $||V^* - V|| \leq \epsilon$. For any $\epsilon > 0$, value iteration converges to an ϵ-optimal value function after a finite number of iterations. A policy $\pi : \mathcal{B} \rightarrow A$ can be extracted from the value function as follows:

$$\pi(b) = \arg\max_{a\in A} \left\{ r(b,a) + \beta \sum_{o\in O} Pr(o|b,a) V(b_o^a) \right\}. \tag{3}$$

Because the dynamic-programming operator is performed on a continuous space of belief states, it is not obvious that it can be computed exactly. However, Smallwood and Sondik (1973) proved that the dynamic-programming operator 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 follows:

$$V(b) = \max_{v^i\in\mathcal{V}} \sum_{s\in S} b(s) v^i(s). \tag{4}$$

Moreover, a piecewise-linear and convex value function has a unique and minimal-size set of vectors that represents it. This representation of the value function allows the dynamic programming operator to be computed exactly. We develop our state aggregation algorithm based on the *incremental pruning* (IP) algorithm (Cassandra, Littman, & Zhang 1997). However our technique can be applied directly to more advanced algorithms such as region-based incremental pruning (Feng & Zilberstein 2004).

Like most other algorithms for computing the DP operator, IP essentially perform two tasks. The first is to generate sets of vectors that may be included in the updated value function. The second is to prune these sets of vectors to their minimal size, by removing dominated or extraneous vectors. Of the two tasks, pruning takes by far the most computation time. Cassandra et al. (1997) report that it takes 95% of the computation time needed to solve a benchmark set of examples, and our experimental results are consistent with this. Both tasks – generating and pruning vectors – can benefit from state aggregation. However, the effect of state aggregation on improving the overall efficiency of the DP operator is much greater in the pruning step because it takes most of the computation time. In the rest of this paper, we describe a method of state aggregation that can be performed in the pruning step for non-factored POMDPs. This method provides most of the same benefit as methods that rely on a factored representation. Besides not requiring a factored representation, it is much simpler to implement and has lower overhead.

Incremental pruning In their description of incremental pruning, Cassandra *et al.* (1997) note that the updated value function V' of Equation (2) can be defined as a combination of simpler value functions, as follows:

$$\begin{aligned} V'(b) &= \max_{a\in A} V^a(b) \\ V^a(b) &= \sum_{o\in\mathcal{O}} V^{a,o}(b) \\ V^{a,o}(b) &= \frac{r(b,a)}{|\mathcal{O}|} + \beta Pr(o|b,a) V(b_o^a) \end{aligned}$$

Each of these value functions is piecewise linear and convex, and can be represented by a unique minimum-size set of state-value functions. We use the symbols $\mathcal{V}'$, $\mathcal{V}^a$, and $\mathcal{V}^{a,o}$ to refer to these minimum-size sets.

Using the script letters $\mathcal{U}$ and $\mathcal{W}$ to denote sets of state-value functions, we adopt the following notation to refer to operations on sets of state-value functions. The *cross sum* of two sets of state-value functions, $\mathcal{U}$ and $\mathcal{W}$, is denoted $\mathcal{U} \oplus \mathcal{W} = \{u + w | u \in \mathcal{U}, w \in \mathcal{W}\}$. An operator that takes a set of state-value functions $\mathcal{U}$ and reduces it to its unique minimum form is denoted PRUNE($\mathcal{U}$). Using this notation, the minimum-size sets of state-value functions defined earlier can be computed as follows:

$$\begin{aligned} \mathcal{V}' &= PRUNE\left(\cup_{a\in A} \mathcal{V}^a\right) \\ \mathcal{V}^a &= PRUNE\left(\oplus_{o\in\mathcal{O}} \mathcal{V}^{a,o}\right) \\ \mathcal{V}^{a,o} &= PRUNE\left(\{v^{a,o,i} | v^i \in \mathcal{V}\}\right), \end{aligned}$$

where $v^{a,o,i}$ is the state-value function defined by

$$v^{a,o,i}(s) = \frac{r(s,a)}{|\mathcal{O}|} + \beta \sum_{s'\in S} Pr(o, s'|s, a) v^i(s'), \tag{5}$$

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

$$\mathcal{V}^a = PRUNE(...PRUNE(\mathcal{V}^{a,o_1} \oplus \mathcal{V}^{a,o_2}) \ldots \oplus \mathcal{V}^{a,o_k}).$$

Table 1 summarizes an algorithm, due to White and Lark (White 1991), that 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.

The simplest method of removing dominated vectors is to remove any vector that is pointwise dominated by another vector. A vector, u, is pointwise dominated by another, w, if $u(s) \leq w(s)$ for all $s \in S$. The procedure POINTWISE-DOMINATE in Table 1 performs this operation. Although this method of detecting dominated state-value functions is fast, it cannot detect all dominated state-value functions.

There is a linear programming method that can detect all dominated vectors. Given a vector v and a set of vectors $\mathcal{U}$ that doesn't include v, the linear program in procedure LP-DOMINATE of Table 1 determines whether adding v to $\mathcal{U}$ improves the value function represented by $\mathcal{U}$ for any belief state b. If it does, the variable d optimized by the linear program is the maximum amount by which the value function

```
procedure POINTWISE-DOMINATE(w, U)
for each u ∈ U
    if w(s) ≤ u(s), ∀s ∈ S then return true
return false

procedure LP-DOMINATE(w, U)
solve the following linear program
    variables: d, b(s) ∀s ∈ S
    maximize d
    subject to the constraints
            b · (w − u) ≥ d, ∀u ∈ U
            Σ_{s∈S} b(s) = 1
if d ≥ 0 then return b
else return nil

procedure BEST(b, U)
max ← −∞
for each u ∈ U
    if (b · u > max) or ((b · u = max) and (u <_lex w)) then
      w ← u
      max ← b · u
return w

procedure PRUNE(U)
W ← ∅
while U ≠ ∅
    u ← any element in U
    if POINTWISE-DOMINATE(u, W) = true
      U ← U − {u}
    else
      b ← LP-DOMINATE(u, W)
      if b = nil then
        U ← U − {u}
      else
        w ← BEST(b, U)
        W ← W ∪ {w}
        U ← U − {w}
return W '
```

Table 1: Algorithm $PRUNE$ for pruning set of state-value functions $\mathcal{U}$.

is improved, and b is the belief state that optimizes d. If it does not, that is, if $d \leq 0$, then v is dominated by $\mathcal{U}$.

The $PRUNE$ algorithm summarized in Table 1 uses these two tests for dominated vectors to prune a set of vectors to its minimum size. (The symbol $<_{lex}$ in the pseudocode denotes lexicographic ordering. Its significance in implementing this algorithm was elucidated by Littman (1994)).

State Aggregation

The linear program used to prune vectors has a number of variables equal to the size of the state space of the POMDP. The key idea of the state aggregation method is this. By aggregating states with the same or similar values, the size of the linear programs (i.e., the number of variables) can be decreased. As a result, the efficiency of pruning – and so, of DP – can be significantly improved.

```
procedure DP-UPDATE(V)
for each action a ∈ A
    V^a ← ∅
    for each observation o ∈ O
        V^{a,o} ← PRUNE({v^{a,o,i} | v^i ∈ V})
        V^a ← PRUNE(V^a ∪ V^{a,o})
V' ← PRUNE(∪_{a∈A} V^a)
return V'
```

Table 2: Incremental pruning algorithm.

Algorithm State aggregation is performed by a *partition algorithm* that takes as input a set of state-value functions and creates an aggregate state space for it that only makes the state distinctions relevant for predicting expected value. Because an aggregate state corresponds to a set of underlying states, the cardinality of the aggregate state space can be much less than the cardinality of the original state space. By reducing the effective size of the state space, state aggregation can significantly improve the efficiency of pruning.

Table 3 describes the CREATE-PARTITION algorithm. The algorithm starts with a single aggregate state that includes all states of the POMDP. Gradually, the aggregation is refined by making relevant state distinctions. Each new state distinction splits some aggregate state into smaller aggregate states. An aggregate state needs to be split when the values of its states in a vector are not all the same. We split the aggregate state by first sorting the states in it according to their values, and then inserting elements in the L array to indicate the new splitting point. The algorithm does not backtrack; every state distinction it introduces is necessary and relevant.

The following example will help to explain the algorithm. We introduce a simple representation for a partition of the state space, using two arrays I and L. I is a permutation of the states, and L specifies the grouping of states in I.

Example Let $S = \{0, 1, 2, 3, 4, 5, 6, 7\}$ be the set of states. A partition that contains three aggregate states $S_0 = \{2, 0, 7\}$, $S_1 = \{4, 5, 3, 1\}$, and $S_2 = \{6\}$ can be represented by I and L as follow:

$$\begin{aligned} I &= [2, 0, 7, 4, 5, 3, 1, 6] \\ L &= [-1, 2, 6, 7] \end{aligned}$$

The set I is a permutation of the states. The set L specifies the grouping of states in I. The number of aggregate states is equal to $|L| - 1$. Each element except the last in L specifies the index range in I that forms a group of states, so that:

$$S_i = \{I[L[i] + j] : 1 \leq j < L[i + 1] - L[i]\}$$

Note that index starts at 0. We use $(I, L)_i$ to denote the i-th aggregate state in the partition defined by I and L.

Now suppose a new vector

s	0	1	2	3	4	5	6	7
v	1.0	3.0	1.0	3.1	3.0	3.1	2.0	1.0

is processed. This introduces inconsistencies in the partition S_1, because states 1 and 4 have value 3.0, and states 3 and

```
procedure CREATE-PARTITION(V, α)
I ← [s_0, s_1, ..., s_n]
L ← [−1, n − 1]
for each vector v ∈ V
    L' ← L
    for i = 0 to |L| − 2
        Sort the portion of I in (I, L)_i according to v
        b ← 1
        for j = 2 to L[i + 1] − L[i] − 1
            if |v(I[L[i] + j]) − v(I[L[i] + b])| > α
                insert (L[i] + j − 1) in L' after L[i]
            b ← j
    L ← L'
return (I, L)
```

Table 3: Algorithm for partitioning a state set, S, into a set of aggregate states, represented by (I, L), that only makes relevant state distinctions found in a set of state-value functions, $\mathcal{V}$.

5 have value 3.1. So the algorithm sorts states in S_1 and adds a split point at position 4, so that the new partition is represented as follow:

$$\begin{aligned} I &= [2, 0, 7, 1, 4, 3, 5, 6] \\ L &= [-1, 2, 4, 6, 7] \end{aligned}$$

The CREATE-PARTITION algorithm is called just before any set of vectors is pruned. The pruning step is then performed on the abstract (or aggregate) state space produced by the algorithm.

Complexity The most expensive operation in the algorithm is sorting, which takes time $O(|S_i| \log |S_i|)$ where $|S_i|$ is the size of a partition being sorted. Note that the size of the partition is the reciprocal of the number of partitions, so the sorting time needed to process each vector can be bounded by $O(|S| \log |S|)$. Thus the worst-case running time of the partition algorithm is $O(|\mathcal{V}||S| \log |S|)$. As we can see in the experimental results, this running time is negligible compared to the running time of the DP algorithm.

Approximation

The partition algorithm also takes a second parameter, α, that allows approximation in the state aggregation process. When $\alpha = 0$, an aggregation that preserves exact values is created. When $\alpha > 0$, states with similar but not necessarily the same values are grouped together, and assigned a similar value. In the above example, if α is set to 0.1, then partition S_1 does not need to be split.

We note that approximation error can be bounded in the same way described by Feng and Hansen (2001), who assume a factored representation of the POMDP. Their analysis extends directly to our algorithm. We list below two important theorems regarding the approximation:

Theorem 1 (Error bound) *The error between the current and optimal value function is bounded as follows,*

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

where $\delta = \alpha \cdot (2|\mathcal{O}| + 1)$ denotes the approximation error.

Theorem 2 (Convergence) *For any value function V and $\varepsilon > 0$, there is an N such that for all $n > N$,*

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

Computational Results

Factored problems We first tested our algorithm on problems that have a factored representation, in order to compare its performance to the factored algorithm of Hansen and Feng (2000). We used problems 3, 4 and 5 (denoted HF3, HF4, HF5) from their paper as test problems. Problem 3 is a widget-processing problem originally described in (Draper, Hanks, & Weld 1994). Problem 4 is an artificial worst-case example in the sense that there is little structure in the problem that allows state aggregation. Problem 5 is an artificial best-case example where there are many similar state values, allowing significant state aggregation.

Table 4 shows some timing results. For the three HF examples, it compares one step of value iteration using three different algorithms: our state aggregation algorithm, the state aggregation algorithm that relies on a factored algorithm, and no state aggregation at all. For each problem, the result without state aggregation is shown at the top. Below it is the result of the factored algorithm, and below that is the result of our new state aggregation algorithm. All programs take the same input and compute identical results. They perform the same number of backups and the same number of pruning operations, and they solve the same number of linear programs. The difference between our algorithm and the factored algorithm is that our algorithm uses a much simpler and more general approach to state aggregation that does not rely on a factored representation.

From the timing results for the pruning operation, which is the most expensive step in DP, we can see that our state aggregation algorithm provides the same level of speed-up as the factored algorithm, when state aggregation is possible. Note that the pruning time in the table includes the time used to create an aggregate state space using the CREATE-PARTITION procedure. As we can see, state aggregation usually take less than 1% of the pruning time, and never takes more than 3% in any case. Compared to the factored algorithm, our state aggregation technique is faster in creating a partition, because we use a simpler data structure that requires less overhead.

For the worst-case example, where state aggregation is not possible, our algorithm spends slightly more time on pruning than the other algorithms. The reason for this slight difference is that our aggregation algorithm re-arranges the order of the states in the vectors, and the actual setup of the linear programs (the order of variables and constraints) can affect the solving time.

For the backup step, we see that our algorithm takes about the same amount of time as the algorithm that does not perform state aggregation, and is usually slower than the factored algorithm. This is because our algorithm does not exploit state abstraction in the backup step, whereas the factored algorithm does. However, as we can see from the ta-

Prune Ex.	$\|S\|$	Abs	Timing Results Back	Part	Prune	Total
		-	1.1	-	5.5	6.7
HF3	2^6	7.5	0.7	0.6	3.4	4.0
			1.2	0.0	2.7	4.0
		-	53.5	-	8127.8	8181.9
HF4	2^6	64.0	404.0	1.8	8244.4	8653.2
			54.0	0.6	8747.1	8801.8
		-	7.7	-	146.3	154.2
HF5	2^7	12.8	5.7	3.8	35.3	41.4
			8.3	1.0	29.0	37.6
		-	4.5	-	139.7	144.5
cit	284	44.0	4.6	1.5	67.3	73.7
penta-		-	114.7	-	681.6	796.3
gon	212	44.8	124.4	6.1	293.8	418.3

Table 4: Timing results (in CPU seconds) for one iteration of DP. The column with the heading "Abs" represents the mean number of abstract states created by the aggregate and factored algorithms, which is averaged over all calls to the procedure PRUNE. "Back" represents the time for computing new vectors. "Part" represents the time for partitioning the state space into aggregate states. "Prune" represents the time for pruning vectors using linear program pruning.

ble, 95% or more of the running time of DP is devoted to the pruning step (except for problem HF3, which is about 82%). Therefore, the advantage enjoyed by the factored algorithm in the backup step is not significant in the overall performance of the algorithm. It is interesting to note that for the worst-case example, the factored algorithm spent about 5 times *more* time in the backup step. This is due to the overhead of manipulating the complex decision diagram data structure, which can be significant when there is little or no structure in the problem.

Non-factored problems Next we tested our algorithm on two robot navigation problems, **cit** and **pentagon**, described in (Cassandra, Kaelbling, & Kurien 1996). These two examples cannot be naturally represented in factored form. We can compare the performance of our algorithm only to the basic algorithm that does not use state aggregation, because the factored algorithm cannot be applied to these problems. In the corresponding entries in Table 4, the top line shows the timing results for the basic algorithm and the bottom line shows the results for the new state aggregation algorithm. It shows that the pruning step can benefit from state aggregation, even for problem that do not have a factored representation.

Approximation We point out again that these timing results are for one iteration of the DP algorithm only. Over successive iterations of the algorithm, the degree of abstraction can decrease as the value function is refined. To counteract this, approximation can be introduced to maintain a similar degree of state abstraction. It is also possible to use

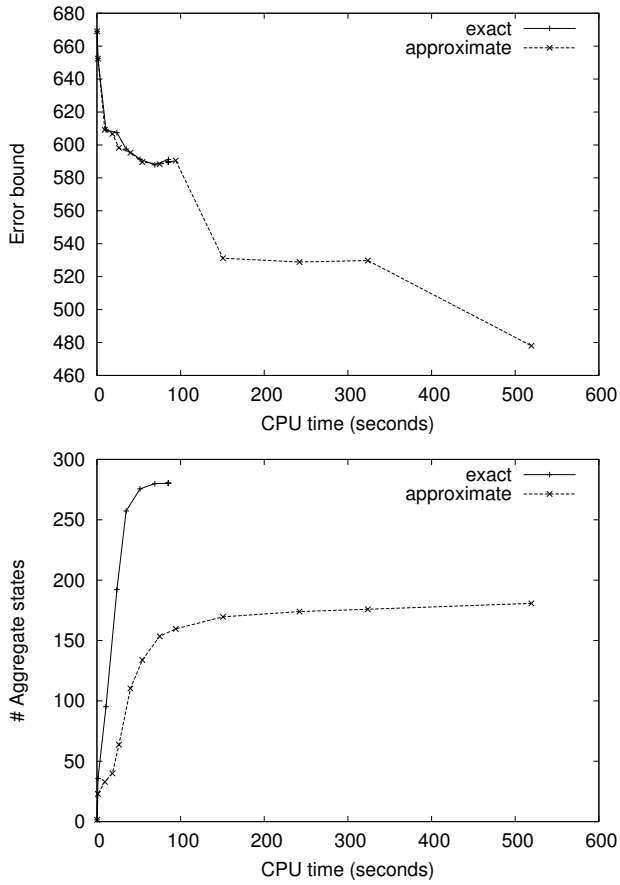

Figure 1: Performance of value iteration using state aggregation, with and without approximation. Each point represents data at the end of an iteration. The top graph shows the error bound at each iteration, and the bottom graph shows the average number of aggregated states at each iteration. Note that value iteration with exact state aggregation did not finish the last iteration after 600 seconds.

approximation in early iterations of value iteration, as of way of speeding convergence.

For example, we implemented a scheme that gradually reduces the approximation error in the state aggregation process, similar to the scheme described in (Feng & Hansen 2001). Figure 1 shows the effect of approximation on the **cit** problem. In the bottom graph, we see that approximation allows more state aggregation than the exact algorithm. This makes it possible to perform more iterations of the DP algorithm, and to reduce the error bound of the value function further than is possible in the same amount of time, using the exact algorithm.

Conclusion

We have presented a simple approach to state aggregation for POMDPs that does not depend on a factored representation, yet retains most of the benefits of the factored algorithms developed before. These benefits include the ability to exploit problem structure in order to reduce the effective size of the state space, and the ability to use approximation

to reduce state space size further, in order to improve the scalability of POMDP algorithms.

Acknowledgments We thank the anonymous reviewers for helpful comments. This work was supported in part by NSF grant IIS-9984952 and IIS-0219606, NASA grant NAG-2-1463, and AFOSR grant F49620-03-1-0090. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the NSF, NASA, or AFOSR.

References

Boutilier, C., and Poole, D. 1996. Computing optimal policies for partially observable Markov decision processes using compact representations. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, 1168–1175.

Cassandra, A. R.; Kaelbling, L. P.; and Kurien, J. A. 1996. Acting under uncertainty: Discrete bayesian models for mobile robot navigation. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Cassandra, A.; Littman, M.; and Zhang, N. 1997. Incremental pruning: A simple, fast, exact method for partially observable markov decision processes. In *Proceedings of the 13th International Conference on Uncertainty in Artificial Intelligence*.

Draper, D.; Hanks, S.; and Weld, D. 1994. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*.

Feng, Z., and Hansen, E. 2001. Approximate planning for factored POMDPs. In *Proceedings of the 6th European Conference on Planning*.

Feng, Z., and Zilberstein, S. 2004. Region-based incremental pruning for POMDPs. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence (UAI-2004)*.

Guestrin, C.; Koller, D.; and Parr, R. 2001. Solving factored POMDPs with linear value functions. In *Proceedings of the IJCAI-2001 workshop on Planning under uncertainty and incomplete information*.

Hansen, E., and Feng, Z. 2000. Dynamic programming for POMDPs using a factored state representation. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling*.

Littman, M. 1994. The witness algorithm: Solving partially observable markov decision processes. Brown university department of computer science technical report cs-94-40.

Smallwood, R., and Sondik, E. 1973. The optimal control of partially observable markov processes over a finite horizon. *Operations Research* 21:1071–1088.

White, C. 1991. A survey of solution techniques for the partially observed markov decision process. *Annals of Operations Research* 32:215–230.