From: AIPS 2000 Proceedings. Copyright © 2000, AAAI (www.aaai.org). All rights reserved.

Dynamic Programming for POMDPs using a Factored State Representation

Eric A. Hansen and Zhengzhu Feng
Computer Science Department and Intelligent Systems Laboratory
Mississippi State University, Mississippi State, MS 39762
{hansen,fengzz}@cs.msstate.edu

Abstract

Contingent planning – constructing a plan in which action selection is contingent on imperfect information received during plan execution – can be formalized as the problem of solving a partially observable Markov decision process (POMDP). Traditional dynamic programming algorithms for POMDPs use a flat state representation that enumerates all possible states and state transitions. By contrast, AI planning algorithms use a factored state representation that supports state abstraction and allows problems with large state spaces to be represented and solved more efficiently. Boutilier and Poole (1996) have recently described how a factored state representation can be exploited by a dynamic programming algorithm for POMDPs. We extend their framework, describe an implementation of it, test its performance, and assess how much this approach improves the computational efficiency of dynamic programming for POMDPs.

Introduction

Many AI planning researchers have adopted Markov decision processes (MDPs) as a framework for decision-theoretic planning. (See (Boutilier *et al.* 1999) for a survey and references.). In doing so, they have also adopted dynamic programming – the most common approach to solving MDPs – as an approach to planning.

Classic dynamic programming algorithms for solving MDPs, including value iteration and policy iteration, 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." Recently, promising extensions of these algorithms have been developed that exploit structure in the representation of an MDP to avoid explicit enumeration of the state space. So far, work in this area has focused on completely observable MDPs and encouraging empirical results have recently been reported (Hoey *et al.* 1999). Boutilier and Poole (1996) have also proposed a value iteration algorithm for partially observable MDPs (POMDPs) that exploits a factored representation. However, their algorithm has not previously been implemented. Boutilier *et al.* (1999) write that "exploiting structured representations of POMDPs is a topic that remains to be explored in depth."

A POMDP models decision making under imperfect and partial state information and supports reasoning about whether to select actions that change the state, provide state information, or both. The problem of planning under uncertainty using information-gathering actions has been considered in the AI community using other frameworks (Draper *et al.* 1994; Onder & Pollack 1999; Majercik & Littman 1999). This paper describes the first implementation of dynamic programming for POMDPs that exploits the same factored representation used by AI planners. We extend the algorithmic framework proposed by Boutilier and Poole and describe value iteration and policy iteration algorithms for POMDPs that adopt this representation. Our results validate this approach and provide a preliminary assessment of how much it can improve the computational efficiency of dynamic programming for POMDPs.

Background

We consider contingent planning problems that are represented in factored form. We assume the relevant properties of a domain are described by a finite set of Boolean state variables, $\mathcal{X} = \{X_1, \ldots, X_n\}$. An assignment of truth values to $\mathcal{X}$ corresponds to a state of the domain. Let $S = 2^{\mathcal{X}}$ denote the set of all possible states. We assume an agent receives (possibly imperfect) state information in the form of a finite set of Boolean observation variables, $\mathcal{Y} = \{Y_1, \ldots, Y_m\}$. An assignment of truth values to $\mathcal{Y}$ corresponds to a particular observation. Let $\mathcal{O} = 2^{\mathcal{Y}}$ denote the set of all possible observations. (The assumption that state and observation variables are Boolean does not limit

Copyright ©2000, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

From: AIPS 2000 Proceedings. Copyright © 2000, AAAI (www.aaai.org). All rights reserved.

the results of this paper. By adopting a suitable encoding, problems with multi-valued variables can be represented using Boolean variables.) We also assume a finite set of actions A.

We briefly review the traditional approach to solving POMDPs before describing how to generalize it to exploit a factored representation.

POMDPs For the class of planning problems we consider, the relationship between an agent and its environment is modeled as a discrete-time POMDP with a finite set of states, S, a finite set of observations, $\mathcal{O}$, and a finite set of actions, A. Each time period, the environment is in some state $s \in S$, the agent chooses an action $a \in A$ for which it receives an immediate reward with expected value $R^a(s) \in \Re$, 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 $Pr(o|s',a) \in [0,1]$. 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 it is followed by observation o, the successor belief state, denoted b_o^a, is determined by revising each state probability using Bayes' theorem, as follows,

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

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 recast as a completely observable MDP with a continuous, $|S|$-dimensional state space consisting of all possible belief states. In this form, it can be solved by iteration of a *dynamic-programming update* that performs the following "one-step backup" for each belief state b:

$$V'(b) := \max_{a \in A} \left[\sum_{s \in S} b(s) R^a(s) + \beta \sum_{o \in \mathcal{O}} Pr(o|b, a) V(b_o^a) \right]. \quad (1)$$

In words, this says that the value of belief state b is set equal to the immediate reward for taking the best action for b plus the discounted expected value of the resulting belief state b_o^a.

Smallwood and Sondik (1973) prove that this dynamic-programming step 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:

$$V(b) = \max_{0 \le i \le k} \sum_{s \in S} b(s) v^i(s). \quad (2)$$

This representation of the value function allows the dynamic programming update to be computed exactly and several algorithms for this have been developed (*e.g.*, Cassandra *et al.* 1997). Value iteration repeatedly applies the dynamic-programming update to improve a value function until convergence to an ϵ-optimal value function. A policy δ mapping belief states to actions can be extracted from the value function using one-step lookahead, as follows,

$$\delta(b) = \arg\max_{a \in A} \left[\sum_{s \in S} b(s) R^a(s) + \beta \sum_{o \in \mathcal{O}} Pr(o|b, a) V(b_o^a) \right].$$

Because value iteration represents a policy implicitly by a value function and improves the policy by gradually improving the value function, it is said to "search in value function space." Another approach to dynamic programming, called policy iteration, is said to "search in policy space" because it represents a policy explicitly and iteratively improves the policy (Sondik 1978; Hansen 1998)

Planning using factored representations Unlike the MDP model developed in operations research, AI planning research has traditionally used factored representations. In the STRIPS action representation, for example, actions are modeled by their effect on state variables. When an action only affects a few state variables, or when its effects are very regular, the STRIPS description of the transition model of the action can be very compact even if the state space of the planning problem is very large.

The representations used by AI planners were originally developed for classical planners that ignored uncertainty, such as the STRIPS planner. In the past few years, there has been growing interest in generalizing these representations to planning problems that include uncertainty about the effects of actions and/or uncertainty about the problem state. A nonlinear planner that solves planning problems that include both forms of uncertainty is described by Draper

From: AIPS 2000 Proceedings. Copyright © 2000, AAAI (www.aaai.org). All rights reserved.

et al. (1994). Onder and Pollack (1999) describe improvements to this algorithm. Majercik and Littman (1999) describe a satisfiability-based planner for the same class of planning problems. Boutilier and Poole (1996) propose exploiting the factored state representation used by such planners to improve the efficiecy of dynamic programming for POMDPs.

Decision diagrams To represent functions that map states to values (or probabilities), traditional POMDP algorithms use a vector that represents each mapping from state to value independently. Boutilier and Poole (1996) propose using decision trees to compactly represent these functions. By mapping sets of states with the same value to the same leaf of the tree, decision trees can compactly represent the same functions. Boutilier and Poole note that other compact representations are possible, including rules and decision lists, and Hoey *et al.* (1999) have recently shown that *algebraic decision diagrams* (ADDs) can compactly represent the transition probabilities, reward function, value function and policy of an MDP. They describe a value iteration algorithm for completely observable MDPs that uses this representation and report that it outperforms a similar algorithm that uses decision trees to compactly represent an MDP. We also use ADDs in our algorithm for solving factored POMDPs.

Decision diagrams are widely used in VLSI CAD because they make it possible 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, $\mathcal{B}^n \rightarrow \Re$. Figures 1, 2, and 3 include examples of ADDs. An ADD can be more compact than a decision tree because it merges branches that lead to the same value. Efficient packages for manipulating ADDs are available that provide operations such as sum, product, expectation, and others (Somenzi 1998).

Factored representation of POMDP

The advantage of using a factored state representation is that it allows a powerful form of state abstraction in which an abstract state is a partial assignment of truth values to $\mathcal{X}$ and corresponds to a set of possible states in the flat representation of the problem. Using abstraction to ignore properties of the state in contexts where they are irrelevant makes it possible to specify a transition and reward model for actions more compactly, as well as to accelerate the dynamic programming algorithm.

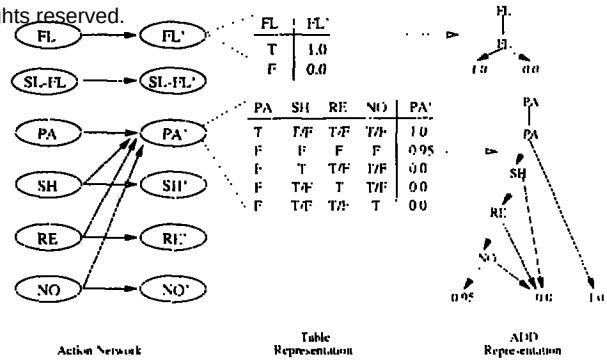

Figure 1: Factored representation of transition probabilities for paint action.

Example As an example of a POMDP with a factored representation, we use a variation of the widget-processing example of Draper *et al.* (1994). Our version of this problem has six Boolean state variables. A widget can be *flawed* (FL), *slightly flawed* (SL-FL), *painted* (PA), *shipped* (SH), and/or *rejected* (RE), and a supervisor can be *notified* (NO) that the widget has been processed. A manufacturing robot has six available actions. It can *inspect*, *repair*, *paint*, *ship*, or *reject* the widget, and it can *notify* a supervisor that the widget has been processed. The objective of the robot is to paint and ship any widget that is not flawed and reject any widget that is flawed. If a widget is only slightly flawed, it can be repaired (removing the slight flaw) and then painted and shipped. (However, the repair and paint actions are only successful with probability 0.95.) The inspect action provides imperfect information about whether a widget is flawed or slightly flawed. There are two observation variables, BAD and SL-BAD. If a widget is flawed, the BAD variable is true with probability 1.0; if it is not flawed, it is true with probablity 0.05. If a widget is slightly flawed, the SL-BAD variable is true with probability 0.95; if it is not slightly flawed, the SL-BAD variable is false with probability 0.0. (But if the widget is flawed, the SL-BAD variable is always false.) After the robot has processed a widget, it notifies the supervisor. It receives a reward of 1.0 for shipping a widget that is unflawed and painted, and a reward of -1.0 for shipping a flawed widget. After notifying the supervisor, the robot receives another widget to process. The probability that the next widget is flawed is 0.3 and the probability that it is slightly flawed is 0.3. All the other state variables are initially false. This example models a simple "assembly line" in which a robot must process an infinite sequence of widgets and its objective is to maximize cumulative discounted reward over an infinite horizon. The discount factor is 0.95.

From: AIPS 2000 Proceedings. Copyright © 2000, AAAI (www.aaai.org). All rights reserved.

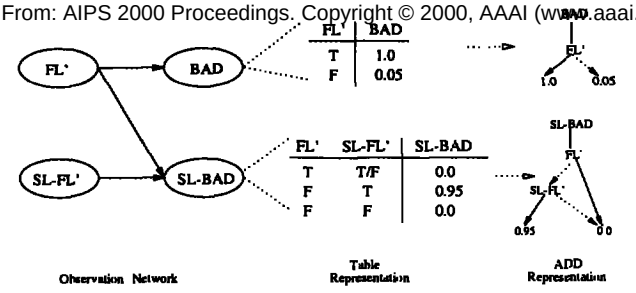

Figure 2: Factored representation of observation probabilities for inspect action.

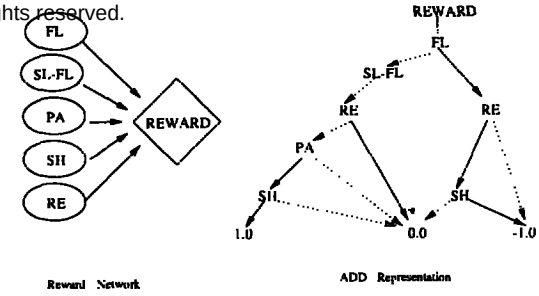

Figure 3: Factored representation of reward function for notify action.

Conditional probability functions A factored representation of a POMDP allows state transition and observation probabilities to be represented compactly. Like Hoey *et al.* (1999), we represent state transition and observation probabilities using two-slice dynamic belief networks. A two-slice dynamic belief network for action a has two sets of variables, one set $\mathcal{X} = \{X_1, ..., X_n\}$ refers to the state before taking action a, and $\mathcal{X}' = \{X'_1, ..., X'_n\}$ refers to the state after. Directed arcs from variables in $\mathcal{X}$ to variables in $\mathcal{X}'$ indicate causal influence. The absence of arcs from variables in $\mathcal{X}$ to variables in $\mathcal{X}'$ reflects variable independence, which allows the state transition probabilities to be represented more compactly. The conditional probability table for each post-action variable X'_i defines a conditional distribution $P^a(X'_i|\mathcal{X})$ over X'_i for each instantiation of its parents. Instead of representing these conditional probabilistics as a table, we follow Hoey *et al.* in representing them using ADDs. (This type of representation is said to exploit context-specific or value independence, in addition to variable independence.) Figure 1 illustrates a dynamic belief network representation of the paint action.

We use a similar factored representation for observation probabilities, as illustrated in Figure 2. Observation probabilities are defined separately for each action. The state variables in the network represent the state that results from the action associated with this network; only the relevant state variables are shown in this network. The conditional probabilities of the observation variables depend on the values of the state variables

Recall that $P^a(X'_i|\mathcal{X})$ denotes the transition probabilities for state variable X'_i after action a. We let $P^a(\mathcal{X}'|\mathcal{X}) = P^a(X'_1|\mathcal{X})...P^a(X'_n|\mathcal{X})$ denote the transition probabilities for all state variables after action a. It is represented by an ADD that is the product of the ADDs representing the conditional probabilities for each state variable. Similarly, we let $P^{a,o}(\mathcal{X}'|\mathcal{X})$ denote the transition probabilities for all state variables after action a and observation o. It is defined in a similar way using the product operation for ADDs. Hoey *et al.* (1999) describe in detail how to use ADDs to compute transition probabilities in the completely observable case. It requires defining both *negative action diagrams* and *dual action diagrams* before performing the product operation on these ADDs. For reasons of space, we refer to that paper for details.

Reward and value functions The reward function for each action, denoted R^a, is also a state-value function and can be represented compactly by an ADD. This is illustrated in Figure 3. Because a piecewise linear and convex value function for a POMDP is repesented by a set of state-value functions $\mathcal{V} = \{v^0, v^1, ..., v^k\}$, it can also be represented compactly by a set of ADDs. In the rest of this paper, we describe how to exploit a factored representation of a piecewise-linear and convex value function for computational speedup.

Dynamic programming update

The central step of dynamic programming algorithms for POMDPs is the step that updates the piecewise linear and convex value function based on a one-step backup for all belief states, as expressed by Equation (1). Several algorithms have been developed that perform this computation. Boutilier and Poole (1996) describe how the simplest of these, called Monahan's algorithm, can exploit a factored representation. In this section, we describe how the most efficient of these algorithms, called incremental pruning, can also exploit a factored representation. We begin with a brief review of incremental pruning before describing how it can be generalized to use a factored representation.

In their description of incremental pruning, Cassandra *et al.* (1997) note that the updated value function V' of Equation (1) can be defined as a combination of

From: AIPS 2000 Proceedings. Copyright © 2000, AAAI (www.aaai.org). All rights reserved.

simpler value functions, as follows:

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

Each of these value functions is piecewise linear and convex. Thus, there is a unique minimum-size set of state-value functions that represents each value function. We use the symbols $\mathcal{V}'$, $\mathcal{V}^a$, and $\mathcal{V}^{a,o}$ to refer to these minimum-size sets. We use the symbol $\mathcal{V}$ to refer to the minimum-size set of state-value functions that represents the current value function V.

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 characterized as follows:

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

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

$$
v^{a,o,i}(s) = \frac{R^a(s)}{|O|} + \beta \sum_{s' \in S} Pr(s', o|s, a) v^i(s'), \quad (3)
$$

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

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

Cassandra *et al.* (1997) describe a *restricted region* generalization of incremental pruning that improves the efficiency of the algorithm further. We do not describe it here because it presents no further complication for use of a factored representation.

Only two steps of the incremental pruning algorithm must be modified to exploit a factored representation: the backup step that creates each $v^{a,o,i}$ (and later combines them using the cross-sum operator) and the pruning step. Below, we describe how to modify these two steps. The rest of the algorithm works exactly the same using a factored representation as it does using a flat representation.

Backup The backup step uses one-step lookahead to create new state-value functions, given a current value function $\mathcal{V}$ and the transition and reward functions of the problem. First, it creates the state-value functions in each set $\mathcal{V}^{a,o}$. For state-value functions represented in flat form, Equation (3) defines how the state-value functions in $\mathcal{V}^{a,o}$ are created. For state-value functions in factored form, the following equation does the same:

$$
v^{a,o,i}(\mathcal{X}) = \frac{R^a(\mathcal{X})}{|O|} + \beta \sum_{\mathcal{X}'} P^{a,o}(\mathcal{X}'|\mathcal{X}) v^i(\mathcal{X}') \quad (4)
$$

In this equation, $v^{a,o,i}$, R^a, $P^{a,o}$, and v^i are represented by ADDs. The operators used in this equation include product of two ADDs and sum of two ADDs. (Multiplication and division of an ADD by a scalar is implemented by first converting the scalar to an ADD representation and then computing the product of two ADDs.) In addition, the symbol $\sum_{\mathcal{X}'}$ denotes an operator that eliminates the primed state variables in the product ADD, $P^{a,o}(\mathcal{X}'|\mathcal{X}) v^i(\mathcal{X}')$, by summing over their values. (In the ADD package, this operator is called existential abstraction.) This operator makes it possible to compute an expected value at the level of state variables instead of states. Once these state-value functions have been created, the cross-sum operator – which simply sums the state-value functions represented by ADDs – is used to create the state-value functions that are eventually contained in $\mathcal{V}'$.

This backup step for POMDPs is very similar to the dynamic programming update step for completely observable MDPs in factored form, as described by Hoey *et al.* (1999). The only difference is that observations and observation probabilities are included in the POMDP version and the maximization operator is not. Given this close correspondence, we refer to their paper for a detailed explanation of how to implement this backup step using a factored representation, such as ADDs. The generalization of this step to POMDPs, assuming a decision tree representation of state-value functions, is described by Boutilier and Poole (1996). Boutilier and Poole also suggest that the pruning step of POMDP algorithms can exploit a factored representation, although they do not develop an algorithm for this. To complement their paper, we proceed to discuss this pruning step at length.

Prune The operator $PRUNE(.)$ takes a set of state-value functions as input and removes dominated state-value functions from it, that is, state-value functions whose removal does not change the belief-value function represented by the set of state-value functions.

The simplest method of removing dominated state-value functions is to remove any state-value function

From: AIPS 2000 Proceedings. Copyright © 2000, AAAI (www.aaai.org). All rights reserved.

```
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 for pruning a set of state-value functions represented by vectors.

that is pointwise dominated by another state-value function. A state-value function, 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 state-value functions. Given a state-value function v and a set of state-value functions $\mathcal{U}$ 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 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 $\mathcal{U}$.

Table 1 summarizes an algorithm, due to White and Lark (White 1991), that uses these two tests for dominated state-value functions to prune a set of state-value functions to its minimum size. (The symbol $<_{lex}$ in the pseudocode denotes lexicographic ordering. Its significance in implementing this algorithm was elucidated by Littman (1994).) To use this algorithm to prune a set of state-value functions represented in factored form, we first perform the pre-processing step summarized in Table 2. It takes as input a set of state-value functions represented in factored form and creates an abstract state space for it that only makes the state distinctions relevant for predicting expected value. Because an abstract state corresponds to a set of underlying states, the cardinality of the abstract state space can be much less than the cardinality of the original state space. By reducing the effective size of the state space, state abstraction can significantly improve the efficiency of pruning because the complexity of the three subroutines used in pruning – POINTWISE-DOMINATE, LP-DOMINATE, and BEST – is a function of the size of the state space. (In the case of the linear programming test for dominated state value functions, state abstraction reduces the number of variables in the linear programs.)

In the pseudocode for CREATE-PARTITION, R denotes a set of abstract states. Initially, R contains a single abstract state that corresponds to the entire state set of the problem. Gradually, R is refined by making relevant state distinctions. Each new state distinction splits some abstract state into two abstract states. The algorithm does not backtrack; every state distinction it introduces is necessary and relevant.

Each state-value function represented by an ADD defines an abstraction of the state space where the number of abstract states is equal to the number of leaves of the ADD, and each abstract state represents a set of underlying states with the same value. Given a set of state-value functions (represented by ADDs), the algorithm summarized in Table 2 creates a set of abstract states that is consistent with the abstractions of the state space created by each state-value function. This means that whenever two states are mapped to different values by the same state-value function, they must belong to different abstract states in R.

From: AIPS 2000 Proceedings. Copyright © 2000, AAAI (www.aaai.org). All rights reserved.

```
procedure CREATE-PARTITION(V)
    R ← {S}
    for each state-value function v ∈ V
        T ← set of abstract states defined by v
        for each abstract state t ∈ T
            for each abstract state r ∈ R
                if (t ∩ r = ∅) or (r ⊆ t) then do nothing
                else if t ⊂ r then
                    R ← R − {r}
                    R ← R ∪ {t} ∪ {r − t}
                    exit innermost for loop
                else
                    R ← R − {r}
                    R ← R ∪ {r − t} ∪ {r ∩ t}
                    T ← T ∪ {t − r}
                    exit innermost for loop
    return R
```

Table 2: Algorithm for partitioning a state set, S, into a set of abstract states, R, that only makes relevant state distinctions found in a set of factored state-value functions, $\mathcal{V}$.

Table 2 describes the CREATE-PARTITION algorithm as if the abstract states it manipulates correspond to sets of underlying states, and conceptually this is true. But in our implementation, we represent abstract states by binary decision diagrams (BDDs) that partition the state set into two sets of states, those that map to 1 and those that map to 0. We define the set of underlying states that corresponds to the abstract state represented by a BDD d as follows:

$$S_d = \{s | d(s) = 1, s \in S\}.$$

This representation of abstract states allows Boolean set operations provided by the ADD package to be used in implementing the algorithm for partitioning the state space into abstract states. We use the logical AND operator, where

$$d_1 \& d_2(s) = 1 \text{ if and only if } d_1(s) = 1 \text{ and } d_2(s) = 1,$$

and the logical complement operator, where

$$\bar{d_1}(s) = 1 \text{ if an only if } d_1(s) = 0,$$

to define the following operations and relations on sets of underlying states corresponding to abstract states:

$$\begin{aligned} S_{d_1} \cap S_{d_2} &= S_{d_1 \& d_2} \\ S_{d_1} - S_{d_2} &= S_{d_1 \& \bar{d_2}} \\ S_{d_1} \subseteq S_{d_2} &\text{ if } d_1 \& d_2 = d_1 \\ S_{d_1} \cap S_{d_2} = \emptyset &\text{ if } d_1 \& d_2 = 0 \end{aligned}$$

Besides creating an abstract state space, our implementation of this pre-processing step returns a vector representation of the set of state-value functions, where the size of each vector is equal to the number of abstract states. (Vectors defined over abstract states can be much smaller than vectors defined over the underlying states; so this is *not* a flat representation of the set of state-value functions.) We do this because the pruning algorithm runs *much* faster when state-value functions are represented as vectors than when they are represented as ADDs. As long as the vectors are defined for abstract states, both the vector and ADD representations are equally compact. (This temporary change of representation also has the benefit of allowing the flat and factored algorithms to use the same code for pruning sets of state-value functions. Thus, the differences in running time reported later are not clouded by implementation differences and accurately reflect the benefits of state abstraction.)

The algorithm for creating an abstract state space summarized in Table 2 has a worst-case complexity of $|\mathcal{V}||S|^2$, although it runs much faster when there is state abstraction. (It is easy to imagine a faster heuristic algorithm that finds a useful, though not necessarily best, state abstraction. For the examples we have tested so far, we have not found this necessary.)

Value and policy iteration

Value iteration for POMDPs consists of repeatedly applying the dynamic programming update. We have also used a factored state representation in implementing a policy iteration algorithm for POMDPs that represents a policy as a finite-state controller (Hansen 1998). By interleaving the dynamic programming update with a policy evaluation step, this algorithm can converge in fewer iterations than value iteration. The policy evaluation step of the algorithm exploits a factored state representation by using a successive approximation algorithm similar to that described by Boutilier *et al.* (1995) to evaluate a controller.

For infinite-horizon POMDPs, the error bound of a value function is computed by solving a set of linear programs. Computation of the error bound can also be accelerated by exploiting a factored state representation, using the same algorithm for creating an abstract state space summarized in Table 2. Space limitations preclude us from describing this here. But we note that the time it takes to compute the error bound is very small, and almost negligible, compared to the time it takes to compute the dynamic programming update.

For the widget processing example, value iteration converges to an optimal belief-value function consisting of 33 state-value functions and policy iteration

From: AIPS 2000 Proceedings. Copyright © 2000, AAAI (www.aaai.org). All rights reserved.

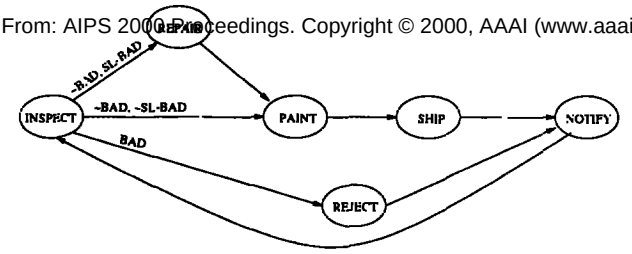

Figure 4: Optimal controller for widget-processing problem (showing only recurrent nodes).

converges to an optimal finite-state controller with 33 nodes. Figure 4 shows the recurrent nodes of the controller, that is, the nodes that are reachable after a widget is processed. The other 27 nodes are transient nodes that are only useful for initial probability distributions that differ from those that characterize the initial condition of the widget. The controller in Figure 4 represents an optimal assembly-line plan for the robot.

Test results

We compared the performance of our implementations of incremental pruning using a factored and a flat representation. Both implementations are identical except that the factored algorithm uses ADDs to compactly represent transition and observation probabilities and state-value functions, while the flat algorithm uses vectors and explicitly represents all states.

Table 3 compares one iteration of incremental pruning using the factored and flat representations, for seven different POMDPs. The results for the flat representation are shown above the results for the factored representation. Both algorithms have the same input and compute identical results. Both perform the same number of backups, the same number of pruning operations, the same number of linear programs, etc. They only differ in the data structures they use.

The procedure PRUNE is invoked multiple times by incremental pruning. Each time it is invoked by the factored algorithm, an abstract state space is created by the procedure CREATE-PARTITION. The column with the heading "Abs" indicates the average cardinality of the abstract state sets that are created. In general, the higher the degree of abstraction for a problem, the faster the factored algorithm runs relative to the flat algorithm. The relationship between degree of abstraction and computational speedup is complicated by other factors, however, including the cardinality of the set of state-value functions and the number of observations. We have broken down the timing results to help illustrate the effect of these various factors. (The total time is slightly larger than the sum of the times for all the operations; this reflects a slight overhead in the algorithm for setting up the computation.)

Problem 1 in Table 3 is the coffee problem used as an illustration by Boutilier and Poole (1996). Problem 3 is the widget-processing problem used as an illustration in this paper. The other problems are various synthetic POMDPs created for the purpose of testing. Problem 4 illustrates the worst-case behavior of the algorithm when there is no state abstraction. Problem 2 illustrates the performance of the algorithm where there is a very modest degree of state abstraction. Problems 5, 6, and 7 illustrate the performance of the algorithm when there is a high degree of state abstraction; they represent close to the best-case behavior of the algorithm for problems of their size. As problem size increases, that is, as the cardinality of S, $\mathcal{O}$, and $\mathcal{V}$ increases, the potential speedup from using a factored representation increases as well.

Two steps of the incremental pruning algorithm exploit a factored representation: the backup step and the pruning step. The table breaks down the timing results for each step. Within each step, it times each operation that uses a factored representation separately. This makes it possible to determine the relative contribution of each operation to the running time of the algorithm, as well as the relative performance of the operations using the factored and flat representations.

The backup step for POMDPs is similar to the dynamic programming update for completely observable MDPs represented in factored form. Like Hoey *et al.* (1999), we found that its worst-case overhead can cause a factored backup to run up to ten times slower than a flat backup. This is illustrated by problem 4. But also like Hoey *et al.*, we found that factored backups can run much faster than flat backups when there is sufficient state abstraction. This is illustrated by problems 6 and 7, in particular.

For most POMDPs, and for all POMDPs that are difficult to solve, most of the running time of dynamic programming is spent pruning sets of state-value functions. In our implementation, the only overhead for the factored pruning algorithm is the overhead for creating an abstract state space. Empirically, we found that this overhead is quite small relative to the running time of the rest of the pruning algorithm. Although it tends to grow with the size of the abstract state space, problem 4, our worst-case example, does not illustrate the worst-case overhead. This is because the procedure CREATE-PARTITION has a useful and reasonable optimization; as soon as the cardinality of the abstract state space is equal to the cardinality of the underlying state space, it terminates. For CREATE-

From: AIPS 2000 Proceedings. Copyright © 2000, AAAI (www.aaai.org). All rights reserved.

#	Problem characteristics $\|S\|$	$\|A\|$	$\|\mathcal{O}\|$	Solution characteristics $\|\mathcal{V}\|$	$\|\mathcal{V}'\|$	Abs	Timing Results Backup New	Xsum	Prune Part	Ptw	LP	Best	Total
						-	0.4	0.08	-	1.96	480.6	15.36	498.7
1	2^5	3	2^1	91	229	14.0	1.1	0.92	9.8	2.53	343.7	7.50	366.3
						-	0.2	0.19	-	23.55	3640.3	166.73	3831.8
2	2^5	6	2^2	25	142	20.9	0.2	0.48	8.6	30.11	3471.4	102.54	3615.3
						-	1.1	0.00	-	0.02	5.4	0.14	6.7
3	2^6	6	2^2	32	30	7.5	0.6	0.02	0.6	0.01	2.7	0.07	4.0
						-	53.4	0.09	-	17.97	7957.8	152.02	8181.9
4	2^6	5	2^2	521	2539	64.0	395.6	8.39	1.8	17.64	8072.1	152.79	8653.2
						-	7.5	0.12	-	0.30	133.0	12.94	154.2
5	2^7	8	2^2	42	42	12.8	5.6	0.15	3.8	0.22	30.5	0.92	41.4
						-	4113.1	1.41	-	2.44	8099.1	324.89	12546.1
6	2^{10}	11	2^3	198	457	34.6	225.3	0.25	63.4	2.40	641.8	19.82	959.7
						-	132.5	0.24	-	0.10	622.5	25.97	782.0
7	2^{10}	11	2^4	4	255	8.9	3.0	0.06	1.0	0.06	22.4	0.52	27.0

Table 3: Timing results (in CPU seconds) for an iteration of the dynamic programming update using incremental pruning. Results for the flat representation are shown above results for the factored representation. The column with the heading "Abs" represents the mean number of abstract states, which is averaged over all calls to the procedure PRUNE. "New" represents the time for computing Equation (4) (for the factored case) and (3) (for the flat case) to create all $v^{a,o,j}$. "Xsum" represents the time for computing cross sums. "Part" represents the time for partitioning the state space into abstract states. "Ptw" represents the time for pointwise dominance pruning. "LP" represents the time for linear program pruning. "Best" represents the time for the procedure BEST.

PARTITION to incur its worst-case overhead. in other words. there must be at least some state abstraction. But this, in turn. can offset the overhead.

For problem 3, there is an apparent anomaly in the timing of the pointwise dominance test; it is slightly slower using a factored representation. The explanation is that the algorithm for detecting pointwise dominance is optimized to terminate as soon as one vector does not dominate the other for some state. Thus, the order of states can affect the timing of this algorithm. although it does so by chance. Because abstract states can be ordered differently than flat states, the running time of the pointwise dominance test is not faster in every case, using a factored representation.

Our results show that exploiting a factored representation for state abstraction can significantly accelerate the pruning algorithm. It can reduce the running time of the pointwise dominance test by decreasing the number of state values that are compared (although our results do not illustrate this well); it can reduce the running time of linear programming by decreasing the number of variables in the linear programs; and it can reduce the running time of the procedure BEST by decreasing the size of the vectors for which a dot product is computed. Because most of the running time of dynamic programming is spent pruning sets of state-value functions, especially using linear programming, incremental pruning using a factored representation can run significantly faster than using a flat representation. Moreover, it incurs very little overhead, even in the worst case. Our results suggest that the worst-case overhead for a factored dynamic programming algorithm for POMDPs is considerably less than the worst-case overhead for a factored dynamic programming algorithm for completely observable MDPs (Hoey *et al.* 1999). The backup step for both algorithms may incur the same overhead. However, the running time of dynamic programming for POMDPs is taken up primarily by pruning sets of state-value functions. This step incurs very little overhead, and thus the combined overhead for the POMDP algorithm can be a much smaller fraction of the algorithm's running time.

Discussion

We have described an implementation of dynamic programming for POMDPs that exploits a factored state representation. It is based on, and extends, a framework proposed by Boutilier and Poole (1996). We extend their framework in the following ways; we use ADDs instead of decision trees, we show how the incremental pruning algorithm (the fastest algorithm for computing the dynamic programming update) can exploit a factored representation, and we describe an algorithm for creating an abstract state space that accelerates the pruning step of dynamic programming. Our

From: AIPS 2000 Proceedings. Copyright © 2000, AAAI (www.aaai.org). All rights reserved.

tion can significantly speed up dynamic programming in the best case and incurs little overhead in the worst case.

This positive result must be placed in perspective, however. The approach we have described addresses POMDP difficulty that is due to the size of the state space. For completely observable MDPs, the size of the state space is the principal source of problem difficulty. For POMDPs, it is not. Although the size of the state space contributes to POMDP difficulty, the size of the value function representing a solution to a POMDP can have a larger influence on problem difficulty. A POMDP with a large state space may have a small optimal value function that can be found quickly. By contrast, a POMDP with a small state space may have a large or unbounded value function that is computationally prohibitive to compute. Although the framework we have described for exploiting a factored state representation can be very beneficial, it does not address the aspect of POMDP difficulty that relates to the size of the value function.

Boutilier and Poole (1996) propose a related approach to approximation that uses a factored representation of state-value functions to represent bounds on state values. By ignoring state distinctions that have little effect on expected value, this approach can reduce the size of the value function computed by dynamic programming. We plan to experiment with this approach to approximation in the near future. We will also explore the possibility of exploiting the factored representation of observations to achieve additional computational speedup.

Acknowledgements We thank the anonymous reviewers for helpful comments.

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. International Conference on Computer-Aided Design, 188–191, IEEE.

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *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), 1104-1111, Montreal, Canada.

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), 1168–1175. Portland, OR.

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), 54–61, Providence, RI.

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 (AIPS-94), 31–36.

Hansen, E. 1998. Solving POMDPs by searching in policy space. In Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98), 211 219. Madison, WI.

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), Stockholm, Sweden.

Littman, M.L. 1994. The Witness algorithm: Solving partially observable Markov decision processes. Brown University Department of Computer Science Technical Report CS-94-40.

Majercik, S.M. and Littman, M.L. 1999. Contingent planning under uncertainty via stochastic satisfiability. In Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99), 549–556, Orlando, FL.

Onder, N. and Pollack, M.E. 1999. Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms. In Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99), 577 – 584, Orlando, FL.

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

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

Sondik, E.J. 1978. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research* 26:282–304.

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