Incremental Policy Generation for Finite-Horizon DEC-POMDPs

Chistopher Amato
Department of Computer Science
University of Massachusetts
Amherst, MA 01003 USA
camato@cs.umass.edu

Jilles Steeve Dibangoye
Laval University
Greyc-CNRS UMR 6072
University of Caen Basse-Normandie
gdibango@info.unicaen.fr

Shlomo Zilberstein
Department of Computer Science
University of Massachusetts
Amherst, MA 01003 USA
shlomo@cs.umass.edu

Abstract

Solving multiagent planning problems modeled as DEC-POMDPs is an important challenge. These models are often solved by using dynamic programming, but the high resource usage of current approaches results in limited scalability. To improve the efficiency of dynamic programming algorithms, we propose a new backup algorithm that is based on a reachability analysis of the state space. This method, which we call incremental policy generation, can be used to produce an optimal solution for any possible initial state or further scalability can be achieved by making use of a known start state. When incorporated into the optimal dynamic programming algorithm, our experiments show that planning horizon can be increased due to a marked reduction in resource consumption. This approach also fits nicely with approximate dynamic programming algorithms. To demonstrate this, we incorporate it into the state-of-the-art PBIP algorithm and show significant performance gains. The results suggest that the performance of other dynamic programming algorithms for DEC-POMDPs could be similarly improved by integrating the incremental policy generation approach.

Introduction

Planning under uncertainty is an important and growing area of AI. In these problems, agents must choose a plan of action based on partial or uncertain information about the world. Due to stochastic actions and noisy sensors, agents must reason about many possible outcomes and the uncertainty surrounding them. A common model used to deal with this type of uncertainty is the partially observable Markov decision process (POMDP). When multiple cooperative agents are present, each agent must also reason about the choices of the others and how they may affect the environment. These team planning problems where each agent must choose actions based solely on local information can be modeled as decentralized POMDPs (DEC-POMDPs).

While several algorithms have been developed for DEC-POMDPs, both optimal and approximate algorithms suffer from a lack of scalability. Many of these algorithms use dynamic programming to build up a set of possible policies from the last step until the first. This is accomplished by "backing up" the possible policies at each step and pruning those that have lower value for all states of the domain and all possible policies of the other agents. Unlike the POMDP case, the current dynamic programming used for DEC-POMDPs exhaustively generates all $t+1$ step policies given the set of t step policies. This is very inefficient, often leading to a majority of policies being pruned after they are generated. Because many unnecessary policies are generated, resources are quickly exhausted, causing only small problems and planning horizons to be solved optimally.

To combat this lack of scalability, we propose a method to perform more efficient dynamic programming by generating policies based on a state space reachability analysis. This method generates policies for each agent based on those that are useful for each given action and observation. The action taken and observation seen may limit the possible next states of the system no matter what actions the other agents choose, allowing only policies that are useful for these possible states to be retained. This approach may allow a smaller number of policies to be generated, while maintaining optimality. Because solutions are built up for each action and observation, we call our method *incremental policy generation*. The incremental nature of our algorithm allows a larger number of dynamic programming steps to be completed and thus, can handle larger plan horizons.

Incremental policy generation is a general approach that can be applied to any DEC-POMDP algorithm that performs dynamic programming backups. These include optimal algorithms for finite (Hansen, Bernstein, and Zilberstein 2004; Boularias and Chaib-draa 2008) and infinite-horizon (Bernstein et al. 2009) DEC-POMDPs as well as many approximate algorithms such as PBDP (Szer and Charpillet 2006), indefinite-horizon dynamic programming (Amato and Zilberstein 2009) and all algorithms based on MBDP (Seuken and Zilberstein 2007b) (e.g. IMBDP (Seuken and Zilberstein 2007a), MBDP-OC (Carlin and Zilberstein 2008) and PBIP (Dibangoye, Mouaddib, and Chaib-draa 2009)). All of these algorithms can be made more efficient by incorporating incremental policy generation.

The remainder of the paper is organized as follows. We begin with background on the DEC-POMDP model and relevant previous work. We then describe incremental policy generation and prove that it allows an optimal finite-horizon solution to be produced with and without start state information. We also show that the approach can be used in an approximate context, specifically with the leading algorithm

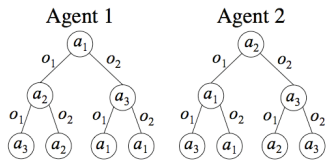

Figure 1: Example of horizon 3 policy trees for two agents.

PBIP. Lastly, we examine the performance of these algorithms on a set of test problems. These experiments show that much better scalability can be achieved with incremental policy generation, allowing larger horizons to be solved than the current optimal and approximate dynamic programming algorithms. When compared with the leading optimal approaches that do not use dynamic programming, increased scalability is also demonstrated for several test cases.

Background

We first describe the DEC-POMDP model and discuss modeling solutions for finite-horizon problems as policy trees.

The DEC-POMDP model

A DEC-POMDP is a general multi-agent decision-theoretic model for planning in partially observable environments. It is defined as: $\langle I, S, \{A_i\}, P, R, \{\Omega_i\}, O\rangle$ with: I, a finite set of agents, S, a finite set of states with designated initial state distribution b_0, A_i, a finite set of actions for each agent, i, P, a set of state transition probabilities: $P(s'|s, \vec{a})$, the probability of transitioning from state s to s' when the set of actions $\vec{a} = (a_1, a_2, \cdots, a_{|I|})$ are taken by the agents, R, a reward function: $R(s, \vec{a})$, the immediate reward for being in state s and taking the set of actions $\vec{a}$, Ω_i, a finite set of observations for each agent, i, O, a set of observation probabilities: $O(\vec{o}|s', \vec{a})$, the probability of seeing the set of observations $\vec{o} = (o_1, o_2, \cdots, o_{|I|})$ given the set of actions $\vec{a}$ was taken, which results in state s', and T, a horizon or finite number of steps after which the problem terminates.

A problem unfolds over a series of steps. At each step, every agent chooses an action based on their local observations, resulting in an immediate reward and an observation for each agent. Note that because the state is not directly observed, it may be beneficial for an agent to remember its observation history. A *local policy* for an agent is a mapping from local observation histories to actions while a *joint policy* is a set of local policies, one for each agent in the problem. The goal is to maximize the total cumulative reward until the horizon is reached, beginning at the given initial state distribution. Seuken and Zilberstein 2008 provide a more thorough introduction to DEC-POMDPs.

Policy trees

To represent policies for finite-horizon DEC-POMDPs, conditional plans represented as a set of local policy trees can be used. These trees are the same as those employed for POMDPs except there is now a policy tree for each agent. An agent's policy tree can be defined recursively. The tree begins with an action at the root and a subtree is defined for each observation. This continues until the horizon of the problem is achieved at the leaf nodes. Thus, the agent's choice of actions is defined by a path through the tree that depends on the observations that it sees. An example of local policy trees for two agents is given in Figure 1.

The joint policy tree is defined by a set of local policy trees, one for each agent, and can be evaluated by summing the rewards at each step weighted by the likelihood of transitioning to a given state and observing a given set of observations. That is, for two agents, the value of trees q_1 and q_2 while starting at state s is given recursively by:

$$V(q_1, q_2, s) = R(a_{q_1}, a_{q_2}, s) + \sum_{s'} P(s'|a_{q_1}, a_{q_2}, s) \sum_{o_1, o_2} P(o_1, o_2|a_{q_1}, a_{q_2}, s') V(q_1^{o_1}, q_2^{o_2}, s')$$

where a_{q_1} and a_{q_2} are the actions defined at the root of trees q_1 and q_2, while $q_1^{o_1}$ and $q_2^{o_2}$ are the subtrees of q_1 and q_2 that are visited after o_1 and o_2 respectively, have been seen.

For agent i's given tree q_i and variables ϵ and $x(q_{-i}, s)$
Maximizes ϵ, given: $\forall\ \hat{q}_i$

$$\sum_{q_{-i},s} x(q_{-i}, s) V(\hat{q}_i, q_{-i}, s) + \epsilon \leq \sum_{q_{-i},s} x(q_{-i}, s) V(q_i, q_{-i}, s)$$

$$\sum_{q_{-i},s} x(q_{-i}, s) = 1 \text{ and } \forall q_{-i}, s\ x(q_{-i}, s) \geq 0$$

Table 1: The linear program that determines if agent i tree, q_i is dominated by comparing its value to other trees for that agent, $\hat{q}_i$. The variable $x(q_{-i}, s)$ is a distribution over trees of the other agents and system states.

Related Work

A number of algorithms have been proposed to build either optimal or approximate joint policy trees. We first discuss the optimal dynamic programming algorithm. We then describe some approximate dynamic programming algorithms and some methods that do not use dynamic programming.

Optimal dynamic programming

Hansen et al. (Hansen, Bernstein, and Zilberstein 2004) developed a dynamic programming algorithm to optimally solve a finite-horizon DEC-POMDP. While this algorithm solves partially observable stochastic games, DEC-POMDPs are a subclass in which the agents share a common payoff function. In this algorithm, a set of T-step policy trees, one for each agent, is generated from the bottom up.

That is, on the Tth (and last) step of the problem, each agent will perform just a single action, which can be represented as a 1-step policy tree. All possible actions for each agent are considered and each combination of these 1-step trees is evaluated at each state of the problem. Any action that has lower value than some other action for all states and possible actions of the other agents is then removed, or pruned. The linear program used to determine whether a tree can be pruned is shown in Table 1. Because there is always an alternative with at least equal value, regardless of system state and other agent policies, a tree q_i can be pruned if ϵ is nonpositive. This pruning can be done for all agents until trees are no longer be pruned.

On the next step, all 2-step policy trees are generated. This is done for each agent by an *exhaustive backup* of the current trees. That is, for each action and each resulting observation some 1-step tree is chosen. If an agent has $|Q_i|$ 1-step trees, $|A_i|$ actions, and $|\Omega_i|$ observations, there will be $|A_i||Q_i|^{|\Omega_i|}$ 2-step trees. After this exhaustive generation of next step trees is completed for each agent, pruning is again used to reduce their number. This generation and pruning continues until the given horizon is reached.

Approximate dynamic programming

The major limitation of dynamic programming approaches is the explosion of memory and time requirements as the horizon grows. This occurs because each step requires generating and evaluating all joint policy trees before performing the pruning step. Approximate dynamic programming techniques somewhat mitigate this problem by keeping a fixed number of local policy trees, MAXTREES, for each agent at each step (Seuken and Zilberstein 2007b). This results in a suboptimal, but much more scalable algorithm.

A number of approaches are based on this idea. The first of which, *memory bounded dynamic programming* (MBDP), has linear space complexity, but still requires an exhaustive backup of policies at each step. To combat this drawback, (Seuken and Zilberstein 2007a; Carlin and Zilberstein 2008) suggested reducing the exponential role of local observations. This is accomplished by performing the exhaustive backup only over a small number of local observations. Unfortunately, the set of policy trees generated before the pruning step is still exponential in the number of local observations retained and the solution quality often decreases.

The leading algorithm of this family of memory bounded approaches is *point-based incremental pruning* (PBIP) algorithm. This approach replaces the exhaustive backup with a branch-and-bound search in the space of joint policy trees (Dibangoye, Mouaddib, and Chaib-draa 2009). Like all MBDP-based approaches, the search technique builds $(t+1)$-step joint policy trees for a small set of selected belief states, one joint policy tree for each belief state. Unlike the previous approaches, it computes upper and lower bounds of the partial $(t+1)$-step joint policy trees. With these bounds, it prunes dominated $(t+1)$-step trees at earlier construction stages. Finally, the best T-step joint policy tree with respect to the initial belief state b_0 is returned. Because each subtree generated after taking a joint action and perceiving a joint observation can be any of the t-step joint policy trees from the last iteration, PBIP does not exploit the state reachability.

Other algorithms

Instead of computing the policy trees from the bottom up as is done by dynamic programming methods, they can also be built from the top down. This is the approach of MAA*, which is an optimal algorithm built on heuristic search techniques (Szer, Charpillet, and Zilberstein 2005). More scalable methods of MAA* have also been proposed using the framework of Bayesian games to provide improved heuristics(Oliehoek, Spaan, and Vlassis 2008; Oliehoek, Whiteson, and Spaan 2009). The recent work on using a generalized version of MAA* with clustering (Oliehoek, Whiteson, and Spaan 2009) improves scalability by representing other agent policies more concisely without losing value. As our focus in on bottom-up dynamic programming methods, we do not discuss these approaches further.

Incremental Policy Generation

As mentioned, one of the major bottlenecks of dynamic programming for DEC-POMDPs is the large resource usage of exhaustive backups. If policies for each agent can be generated without first exhaustively generating all next step possibilities, the algorithm would become more scalable. Incremental policy generation provides a method to accomplish this, while maintaining optimality.

Difference with POMDP methods To improve the efficiency of dynamic programming for POMDPs, methods have been developed such as incremental pruning (Cassandra, Littman, and Zhang 1997), which incrementally generates and prunes an agent's polices rather than using exhaustive backups. Unfortunately, these methods cannot be directly extended to DEC-POMDPs. This is because in the decentralized case, the state of the problem and the value of an agent's policy depend on the policies of all agents. Thus, pruning any agent's $(t+1)$-step policy requires knowing all $(t+1)$-step policies for the other agents.

DEC-POMDP approach To solve this problem, we build up the policies for each agent by using a one-step reachability analysis. This allows all other agent policies to be considered, but without actually constructing them exhaustively and only evaluating them at reachable states. After an agent chooses an action and sees an observation, the agent may not know the state of the world, but it can determine which states are possible. For instance, assume an agent has an observation in which a wall is perfectly observed when it is to the left or right. If the agent sees the wall on the right, it may not know the exact state it is in, but it can limit the possibilities to those states with walls on the right. Likewise, in the commonly used two agent tiger domain (Nair et al. 2003), after an agent opens a door, the problem transitions back to the start state. Thus, after an open action is performed, the agent knows the exact state of the world.

So, how can we use this information? One way is to limit the policies generated during dynamic programming. We can first determine which states are possible after an agent chooses an action and sees an observation. Only subtrees that are useful for the resulting states need to be considered after that observation is seen when constructing trees that begin with the given action. To maintain optimality, we consider any first action for the other agents and then prune using the known possible subtrees for the other agents.

Limiting the state

Any agent can calculate the possible states that result from taking an action and seeing an observation. To determine the next state exactly, the agent would generally need to know the probability that the system is currently in any given state and that the other agents will choose each action. Since we do not assume the agents possess this information, the exact

Algorithm 1: Incremental policy generation

input : A set of t-step policy trees for all agents, Q, and an action a
output: A set of $(t+1)$-step policy trees for a given agent, $\hat{Q}_i$
begin
  for *each action, a* **do**
    for *each observation, o* **do**
      $S' \leftarrow possibleStates(a, o)$
      $Q_i^{a,o} \leftarrow usefulTrees(S', Q)$
    $\hat{Q}_i^a \leftarrow \oplus_o Q_i^{a,o}$
  $\hat{Q}_i \leftarrow \cup_a Q_i^a$
  return $\hat{Q}_i$
end

state can only be known in some circumstances, but the set of possible next states can often be limited.

For instance, the probability of a resulting state s' after agent i chooses action a_i and observes o_i is determined by:

$$P(s'|a_i, o_i) = \frac{\sum_{\vec{a}_{-i},\vec{o}_{-i},s} P(\vec{o}|s,\vec{a},s')P(s'|s,\vec{a})P(\vec{a}_{-i},s|a_i)}{P(o_i|a_i)} \tag{1}$$

with $P(\vec{a}_{-i}, s|a_i)$ the probability of the current state and that other agents choose actions $\vec{a}_{-i}$. The normalizing factor is:

$$P(o_i|a_i) = \sum_{\vec{a}_{-i},\vec{o}_{-i},s,s'} P(\vec{o}|s,\vec{a},s')P(s'|s,\vec{a})P(\vec{a}_{-i},s|a_i).$$

To determine all possible next states for a given action and observation, we can assume $P(\vec{a}_{-i}, s|a_i)$ is a uniform distribution and retain any state s' that has a positive probability. We will call the set of possible successor states S'.

Knowing the exact state When $P(o_i|s,\vec{a},s')P(s'|\vec{a},s)$ is constant for the given a_i and o_i, then it does not depend on the state or other agents' actions. Thus, $\sum_{\vec{a}_{-i},s} P(\vec{a}_{-i}, s|a_i) = 1$ and the exact state is known by:

$$P(s'|a_i, o_i) = \frac{P(o_i|s,\vec{a},s')P(s'|s,\vec{a})}{\sum_{s'} P(o_i|s,\vec{a},s')P(s'|s,\vec{a})}$$

Algorithm

Our approach is summarized in Algorithm 1. We describe it from a finite-horizon perspective, but the same approach can be used in infinite-horizon policy iteration (Bernstein et al. 2009). For each agent, i, assume we have a set of horizon t trees Q_i. We can create a set of horizon $t+1$ trees that has each action from the domain as a first action. Assuming we start with action a, we need to decide which subtrees to choose after seeing each observation. Rather than adding all possibilities from Q_i as is done in the exhaustive approach, we only add trees from Q_i that are useful for some possible successor state s' and set of trees of the other agents. That is, agent i's tree q_i is retained if the value of choosing q_i is higher than choosing any other tree for some distribution of other agent trees, q_{-i} and next states of the system s'. This is formulated as: $\forall\ \hat{q}_i$

$$\sum_{q_{-i},s'} x(q_{-i}, s')V(q_i, q_{-i}, s') > \sum_{q_{-i},s'} x(q_{-i}, s')V(\hat{q}_i, q_{-i}, s')$$

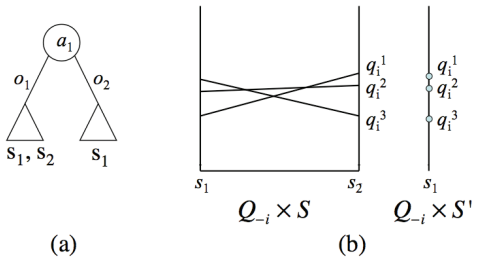

Figure 2: Example of (a) limiting states after actions are chosen and observations seen and (b) pruning with a reduced state set, S'.

Where $x(q_{-i}, s')$ is a distribution over trees of the other agent and successor states, $q_i, \hat{q}_i \in Q_i$, $q_{-i} \in Q_{-i}$ and $s' \in S'$ as described in the previous section.

The solution to this formulation can be found by using the linear program in Table 1, but the new linear program has fewer variables and constraints, therefore making it faster to solve. This is because it uses the smaller set of trees from step t rather than step $t+1$ and $S' \subseteq S$. The method limits the possible subtrees after taking action a and seeing observation o to those that are useful for some distribution over successor states and other agent subtrees. The set of trees that we retain for the given action and observation are t-step trees for agent i, which we will call $Q_i^{a,o}$, leaving off the i for the action and observation to simplify the notation.

After generating all possible sets of trees for each observation with a fixed action, we then build the horizon $t+1$ trees for that action. This is accomplished by creating all possible trees that begin with the fixed action, a, followed by choosing any tree from the set $Q_i^{a,o}$ after o has been observed. The resulting number of trees for agent i at this step is given by $\prod_o |Q_i^{a,o}|$. In contrast, exhaustive generation of trees would produce $|Q_i|^{|\Omega_i|}$ trees. Once all trees for each action have been generated, we take the union of the sets for each action to produce the set of horizon $t+1$ trees for the agent. When we have the $(t+1)$-step trees for all the agents, we are then able to evaluate them and prune those that have lesser value for all initial states and trees of the other agents. This pruning step is exactly the same as that used in Hansen et al.'s dynamic programming algorithm and may further reduce the number of trees retained.

Example An illustration of the incremental policy generation approach is given in Figure 2. The algorithm (a) first determines which states are possible after a given action has been taken and each observation has been seen. After action a_1 is taken and o_1 is seen, states s_1 and s_2 are both possible, but after o_2 is seen, only s_1 is possible. Then, (b) the values of the trees from the previous step (t) are determined for each resulting state and each set of trees of the other agents.

The dimension of the value vectors of the trees is the same as the number of possible policies for the other agents times the number of states in the system, or $Q_{-i} \times S$. To clarify the illustration, we assume there is only one possible policy for the other agents and two states in the domain. As seen in the figure, when both states are considered, all trees for agent i are useful (not dominated) for some possible distribution.

When only state 1 is possible, only q_i^1 is useful. This allows both q_i^2 and q_i^3 to be pruned for this combination of a_1 and o_2, reducing the number of possibilities to 1. Because there are still 3 useful subtrees for the combination of a_1 and o_1, the total number of possible trees starting with action a_1 is 3. This can be contrasted with the 9 trees that are possible using exhaustive generation.

Analysis

We first show that the incremental policy generation approach does not prune any t-step subtree of Q_i that would be part of an undominated $(t+1)$-step policy tree.

Lemma 1. *Any t-step that is not added by incremental policy generation is part of a dominated $(t+1)$-step tree.*

Proof. We will prove this for two agents, but this proof can easily be generalized to any number of agents. We show from agent 1's perspective that if tree q_1 has a subtree which is not included by our incremental policy generation algorithm, then q_1 must be dominated by some distribution of trees $\hat{q}_1$. This is determined by

$$V(q_1, q_2, s) \leq \sum_{\hat{q}_1} x(\hat{q}_1) V(\hat{q}_1, q_2, s) \; \forall q_2, s$$

This proof is based on the fact that in incremental policy generation, all reachable next states and all possible next step policies for the other agents are considered before removing a policy.

Consider the set of trees generated for agent 1 by starting with action a. If observation o_1^j was seen, where $q_1^{o_1^j}$ was not included in set $Q_1^{a_{q_1}, o_1^j}$, then we show that q_1 must be dominated by the set of trees that are identical to q_1 except instead of choosing $q_1^{o_1^j}$, some distribution over trees in $Q_1^{a_{q_1}, o_1^j}$ is chosen. We abbreviate $q_1^{o_1^j}$ by q_1' to simplify notation.

Because q_1' was not included in $Q_1^{a_{q_1}, o_1^j}$ we know there is some distribution of subtrees that dominates it for all subtrees of the other agent and at the subset of states that are possible after choosing action a_{q_1} and observing o_1.

$$V(q_1', q_2', s') \leq \sum_{\hat{q}_1'} x(\hat{q}_1') V(\hat{q}_1', q_2', s') \; \forall \; q_2', s' \qquad (2)$$

Given this distribution $x(\hat{q}_1')$ we can create a probability distribution over subtrees that chooses the trees of $\hat{q}_1'$ when o_1^j is seen, but otherwise chooses the same subtrees that are used by q_1. The value of this tree is given by

$$R(a_{q_1}, a_{q_2}, s) + \sum_{s'} P(s'|a_{q_1}, a_{q_2}, s) \sum_{o_1, o_2} P(o_1, o_2|a_{q_1}, a_{q_2}, s') \sum_{\hat{q}_1'} x(\hat{q}_1', o_1) V(q_1^{o_1}, q_2^{o_2}, s')$$

Because the trees are otherwise the same, from the inequality in Equation 2, we know that

$$R(a_{q_1}, a_{q_2}, s) + \sum_{s'} P(s'|a_{q_1}, a_{q_2}, s) \sum_{o_1, o_2} P(o_1, o_2|a_{q_1}, a_{q_2}, s') V(q_1^{o_1}, q_2^{o_2}, s') \geq R(a_{q_1}, a_{q_2}, s) + \sum_{s'} P(s'|a_{q_1}, a_{q_2}, s) \sum_{o_1, o_2} P(o_1, o_2|a_{q_1}, a_{q_2}, s') \sum_{\hat{q}_1'} x(\hat{q}_1', o_1) V(q_1^{o_1}, q_2^{o_2}, s')$$

This holds for any s and q_2 because Equation 2 holds for any initial state or possible policy tree of the other agent.

This can also be viewed as a distribution of trees, $x(\hat{q}_1)$, one for each different subtree with positive probability in $x(\hat{q}_1')$. Thus, the value of the original tree q_1 is less than or equal to the value of the distribution of trees for all q_2 and s.

$$R(a_{q_1}, a_{q_2}, s) + \sum_{s'} P(s'|a_{q_1}, a_{q_2}, s) \sum_{o_1, o_2} P(o_1, o_2|a_{q_1}, a_{q_2}, s') V(q_1^{o_1}, q_2^{o_2}, s') \leq \sum_{\hat{q}_1} x(\hat{q}_1) \left[R(a_{\hat{q}_1}, a_{q_2}, s) + \sum_{s'} P(s'|a_{\hat{q}_1}, a_{q_2}, s) \sum_{o_1, o_2} P(o_1, o_2|a_{\hat{q}_1}, a_{q_2}, s') V(\hat{q}_1^{o_1}, q_2^{o_2}, s') \right]$$

or $\qquad V(q_1, q_2, s) \leq \sum_{\hat{q}_1} x(\hat{q}_1) V(\hat{q}_1, q_2, s) \; \forall q_2, s$ $\qquad \square$

Theorem 1. *Incremental policy generation returns an optimal solution for any initial state.*

Proof. This proof follows from Hansen et al.'s proof and Lemma 1. Dynamic programming with pruning generates and retains all optimal policies for any given horizon and initial state. Since Lemma 1 shows that incremental policy generation will not remove any policies that would not be pruned, our approach also retains all optimal policies. Thus, by choosing the highest valued policy for any horizon and initial state, our incremental policy generation algorithm provides the optimal finite-horizon solution. $\square$

Incorporating Start State Information

Dynamic programming creates a set of trees for each agent that contains an optimal tuple of trees for any initial state distribution. This results in many trees being generated that are suboptimal for a known initial distribution. To combat this, start state information can be incorporated into the algorithm. We discuss an optimal and an approximate method of including this information.

Optimal algorithm

As already mentioned, any agent i can determine its next possible states $P(s'|a_i, o_i)$ after taking an action a_i and perceiving an observation o_i. This allows the agent to focus its policy generation for the next step over those states only. Although this approach is more scalable than previous dynamic programming methods, further scalability can be achieved by incorporating information from a known start state.

It is well known that the reachable part of the state space can be limited by considering actual agent action sequences (histories) and the initial state information at hand (Szer and Charpillet 2006). However, the only attempt, suggested by Szer and Charpillet that exploits reachability by taking into account the other agents' histories leads to a worst-case complexity that is doubly exponential. We take a different approach, as we only want to determine the possible next states for each agent rather than constructing all possible

trees of the desired horizon, T. Our approach generates possible states for any number of steps, $T-t$, without explicitly generating all horizon T policy trees.

Consider the case when we have already generated the policy trees of height $T-1$ for all agents. We can then use the start state as the only state in S to better limit S' in our incremental policy generation approach. That is, we know that we will begin in b_0, transition to some state s', and choose some height $T-1$ tree. Thus, in Equation 1, $P(\vec{a}_{-i}, s|a_i)$ becomes $b_0(s)P(\vec{a}_{-i}, |a_i)$ and requires using only the known initial state b_0 in addition to the unknown probability of action for the other agents. Then, by considering each initial action of the other agents, we can generate a smaller set of possible next states.

This idea can be extended to any step of dynamic programming. When we have trees of height t, with $t < T-1$ then we must consider all possible action sequences, or histories, of the agents rather than just single actions. We define a k-step history for agent i as $h_i^k = (a_i^1, o_i^1, \cdots, o_i^k, a_i^k)$. The probability of state s^{k+1} resulting after k steps of agent i's history can be defined recursively as: $P(s^{k+1}|h_i^k) \propto$

$$\sum_{\vec{a}_{-i}, \vec{o}_{-i}, s^k} P(o_i^k, \vec{o}_{-i}|s^k, a_i^k, \vec{a}_{-i}, s^{k+1}) P(s^{k+1}|s^k, \vec{a}) P(s^k|h_i^{k-1})$$

where a_i^k and o_i^k are the action taken and observation seen by agent i at the kth step of history h_i^k. The value can be normalized to give the true probability of each state after k steps and for $k=0$, $P(s) = b_0(s)$, the initial state probability.

This approach limits the set of reachable states, S', for agent i given both the initial state information and agent history. This is done for all possible histories of the agent and for all agents. Each agent's policy generation process then follows as described for incremental policy generation. Because the set of all possible histories of any agent is exponential with respect to the length of the histories, this method becomes impractical when $T-t$ is large. To combat this, start state information could be used only when $T-t$ is small and repeated state distributions can be ignored at each level.

Corollary 2. *Our incremental policy generation algorithm with start state information returns an optimal finite-horizon solution for the given initial state distribution.*

Proof. Because we generate all possible next states for each agent using all possible histories for all agents and the given initial state information, this corollary follows from Theorem 1 and Theorem 1 of (Szer and Charpillet 2006). □

Approximate algorithm

To improve the worst-case complexity of our algorithm, it can be incorporated into a memory bounded (MBDP-based) algorithm. MBDP-based approaches (Seuken and Zilberstein 2007b) make use of additional assumptions: (1) the joint state information, *i.e.* joint belief state, is accessible at planning time, although this information is not available to the agents at execution time; (2) only a small number of joint belief states are considered at each decision step. Because all current memory bounded algorithms use a form of exhaustive backup, incremental policy generation can be used to make the backup step more efficient. This can be accomplished by using assumption (1) to provide a set of known states for our incremental policy generation algorithm. That is, given a joint belief state, one can find all the possible next states that are reachable. This can be viewed as a special case of the optimal version described above, with the simplifying assumption that the system state is revealed after $T-t$ steps. Thereafter, one can select the subtrees after step t for agent i, $Q_i^{a,o}$, to generate set Q_i for each agent.

Integrating with PBIP While incremental policy generation could be incorporated into any memory bounded approach, in this paper we chose the point-based incremental pruning (PBIP) algorithm. To do so, we provide PBIP with the set $Q^{\vec{a},\vec{o}} \leftarrow \otimes_i Q_i^{a,o}$ of t-step joint policies that can be executed if we took joint action $\vec{a}$ one step before and any possible joint observation $\vec{o}$ is seen. With this, PBIP is now able to distinguish between subtrees executed after taking a given action and perceiving a given observation. Indeed, it is likely that subtrees that are good candidates for joint action joint observation pairs are useless for another pair.

Experiments

We first provide results comparing incremental policy generation with optimal algorithms and then give results for our approach incorporated into the approximate PBIP algorithm.

Optimal approaches

We first compare incremental policy generation with other state-of-the-art finite-horizon algorithms. These are dynamic programming for POSGs (Hansen, Bernstein, and Zilberstein 2004) and the leading top-down approaches GMAA* (Oliehoek, Spaan, and Vlassis 2008) and clustering GMAA* (Oliehoek, Whiteson, and Spaan 2009), abbreviated as C-GMAA*. These algorithms were tested on the common Box Pushing problem (Seuken and Zilberstein 2007a), the Stochastic Mars Rover problem (Amato and Zilberstein 2009) and a new version of the Meeting in a Grid problem (Bernstein, Hansen, and Zilberstein 2005). This version has a grid that is 3 by 3 rather than 2 by 2, the agents can observe walls in four directions rather than two and they must meet in the top left or bottom right corner rather than in any square. The resulting problem has 81 states, 5 actions and 9 observations. We chose these domains to explore the ability of the algorithms to solve large problems.

The dynamic programming algorithms were run with up to 2GB of memory and 2 hours. We record the number of trees generated by the backup before and after pruning as well as the running time of the algorithm. When the available resources were exhausted during the pruning step, this is denoted with *. The incremental policy generation (IPG) approach using start state information only used the start state for steps when $T-t \leq T/2$ for current tree height t and horizon T. The top-down algorithm results for the Box Pushing domain are taken from (Oliehoek, Whiteson, and Spaan 2009) and the other results are provided courtesy of Matthijs Spaan. Because search is used rather than generating multiple trees, the average size of the Bayesian game used to calculate the heuristic on the final step is provided

Horizon	DP for POSGs	Incremental Generation (IPG)	IPG with Start State	GMAA$^*_{MDP}$	C-GMAA$^*_{MDP}$	Value
Meeting in a 3x3 Grid, $\|S\| = 81$, $\|A_i\| = 5$, $\|\Omega_i\| = 9$						
2	(5) 5 in 5s	5 in <1s	5 in 5s	x	9 <1s	0.000
3	x	5 in 16s	5 in 17s	x	121 <1s	0.133
4	x	40 in 42s	10 in 53s	x	x	0.433
5	x	(25960)* in 2555s	(148) 148,145 in 600s	x	x	0.896
Box Pushing, $\|S\| = 100$, $\|A_i\| = 4$, $\|\Omega_i\| = 5$						
2	(128) 8 in 14s	8 in 2s	(4) 2,3 in 1s	25 in $< 1s$	4 in $< 1s$	17.60
3	x	(320,256) 256 in 1159s	(6) 5,6 in 6s	x	25 in 5s	66.08
4	x	x	(233,239) 233 in 1138s	x	x	98.59
Stochastic Mars Rover, $\|S\| = 256$, $\|A_i\| = 6$, $\|\Omega_i\| = 8$						
2	x	(150, 672)* in 72s	(16,20) 12,15 in 83s	x	1 <1s	5.80
3	x	x	(396, 534)* in 389s	x	4 <1s	9.38
4	x	x	x	x	11.11 in 103s	10.18

Table 2: Size, running time and value produced for each horizon on the test domains. For dynamic programming algorithms the size is given as the number of of policy trees before and after pruning (if different) and only one number is shown if both agent trees are the same size. For top-down approaches the size of the final Bayesian game is provided.

along with the running time. Running times may vary as these algorithms were run on a different computer, but we expect the trends to remain consistent.

The results in Table 2 show that on these problems, incremental policy generation is always more scalable than the previous dynamic programming approach and generally more scalable than the leading top-down approach, C-GMAA*. Dynamic programming (DP) for POSGs can only provide a solution of horizon two for the first two problems and horizon one for the larger Mars Rover problem. GMAA* is similarly limited because the search space quickly becomes intractable as the horizon grows. Because the structure of these problems permits clustering of agent policies, C-GMAA* runs very quickly and is able to improve scalability over GMAA*, especially in the Mars Rover problem. In the other domains, IPG with start state information can reach larger horizons than the other approaches. There is a small overhead of using start state information, but this approach is generally faster and more scalable than the other dynamic programming methods because fewer trees are retained at each step. IPG without start state information is similarly faster and more scalable than the previous dynamic programming algorithm because it is also able to retain fewer trees. These results show that incorporating incremental policy generation greatly improves the performance of dynamic programming, allowing it outperform the leading top-down approach on two of the tree test problems.

Approximate approaches

We now examine the performance increase achieved by incorporating the incremental policy generation approach into the leading approximate algorithm, PBIP. Only PBIP is used because it always produces values at least as high as IMBDP (Seuken and Zilberstein 2007a) and MBDP-OC (Carlin and Zilberstein 2008) and is more scalable than MBDP (Seuken and Zilberstein 2007b). It is worth noting that IPG can also be incorporated into each of these algorithms to improve their efficiency. The same domains as above are used with a choice for MAXTREES fixed at 3 for each algorithm. Due to the stochastic nature of PBIP, each method was run 10 times and the mean values and running times are reported.

Experimental results are shown in Table 3 with PBIP and PBIP with the incremental policy generation approach (termed PBIP-IPG). It can be seen that PBIP is unable to solve the Meeting in a 3 by 3 Grid or Mars Rover problems for many horizons in the allotted time (12 hours). Incorporating IPG allows PBIP to solve these problems for much larger horizons. On the Box Pushing domain, PBIP is able to solve the problem on each horizon tested, but PBIP-IPG can do so in at most half the running time. These results show that while the branch and bound search used by PBIP allows it to be more scalable than MBDP, it still cannot solve problems with a larger number of observations. Incorporating IPG allows these problems to be solved because it uses action and observation information to reduce the number of trees considered at each step. Thus, the exponential affect of the number of observations is reduced by the IPG approach.

Figure 3 shows the running time for different choices of MAXTREES (the number of trees retained at each step of dynamic programming) on the Box Pushing domain with horizon 10. While the running time increases with the number of MAXTREES for both approaches, the time increases more slowly with the IPG approach. As a result, a larger number of MAXTREES can be used by PBIP-IPG. This is due to more efficient backups, which produce fewer horizon $t + 1$ trees for each horizon t tree. These results, together with

Horizon	PBIP	PBIP-IPG	Value
Meeting in a 3x3 Grid, $\|S\| = 81$, $\|A_i\| = 5$, $\|\Omega_i\| = 9$			
10	x	352s	3.85
100	x	3084s	92.12
200	x	13875s	193.39
Box Pushing, $\|S\| = 100$, $\|A_i\| = 4$, $\|\Omega_i\| = 5$			
100	536s	181s	598.40
1000	5068s	2147s	5707.59
2000	10107s	4437s	11392.03
Stochastic Mars Rover, $\|S\| = 256$, $\|A_i\| = 6$, $\|\Omega_i\| = 8$			
2	106s	19s	5.80
10	x	976s	21.18
20	x	14947s	37.81

Table 3: Running time and value produced for each horizon using PBIP with and without incremental policy generation (IPG).

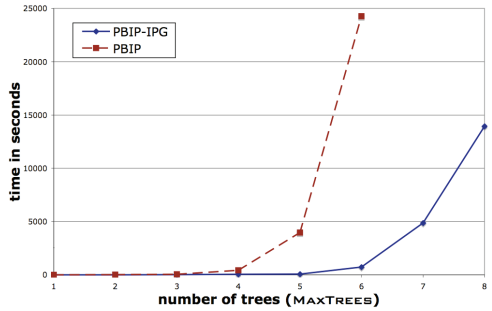

Figure 3: Running times for different values of MAXTREES on the Box Pushing problem with horizon 10.

those from Table 3 show that incorporating the incremental policy generation approach allows improved scalability to larger horizons and a larger number of MAXTREES.

Conclusion

In this paper, we introduced the incremental policy generation approach, which is a more efficient way to perform dynamic programming backups in DEC-POMDPs. This is achieved by using state information from each possible action taken and observation seen to reduce the number of trees considered at each step. We proved that this approach can be used to provide an optimal finite-horizon solution and showed that this results in an algorithm that is faster and can scale to larger horizons than the current dynamic programming approach. We also showed that in two of three test domains, it solves problems with larger horizon than the leading optimal top-down approach, clustering GMAA*.

Incremental policy generation is a very general approach that can improve the efficiency of any DEC-POMDP algorithm that uses dynamic programming. To test this generality, we also incorporated our approach into the leading approximate algorithm, PBIP. The results show that resource usage is significantly reduced, allowing larger horizons to be solved and more trees to be retained at each step.

Because incremental policy generation uses state information from the actions and observations, it should work well when a small number of states are possible for each action and observation. In contrast, clustering GMAA* uses the value of agent policies to cluster action and observation histories. Since these approaches use different forms of problem structure, it may be possible to combine them either by producing more focused histories when making use of start state information or better heuristic policies for use with MBDP-based approaches. Other work has also been done to compress policies rather than agent histories, improving the efficiency of the linear program used for pruning (Boularias and Chaib-draa 2008). By also incorporating incremental policy generation, this combination of techniques could be applied to further scale up dynamic programming algorithms. Lastly, we plan to investigate the performance improvements achieved by incorporating incremental policy generation into infinite-horizon DEC-POMDP algorithms.

Acknowledgments

This work was supported in part by the Air Force Office of Scientific Research under Grant No. FA9550-08-1-0181 and by the National Science Foundation under Grant No. IIS-0812149.

References

Amato, C., and Zilberstein, S. 2009. Achieving goals in decentralized POMDPs. In *Proc. of the Eighth Int. Joint Conf. on Autonomous Agents and Multiagent Systems*.

Bernstein, D. S.; Amato, C.; Hansen, E. A.; and Zilberstein, S. 2009. Policy iteration for decentralized control of Markov decision processes. *Journal of AI Research* 34.

Bernstein, D. S.; Hansen, E.; and Zilberstein, S. 2005. Bounded policy iteration for decentralized POMDPs. In *Proc. of the Nineteenth Int. Joint Conf. on Artificial Intelligence*.

Boularias, A., and Chaib-draa, B. 2008. Exact dynamic programming for decentralized POMDPs with lossless policy compression. In *Proc. of the Eighteenth Int. Conf. on Automated Planning and Scheduling*.

Carlin, A., and Zilberstein, S. 2008. Value-based observation compression for DEC-POMDPs. In *Proc. of the Seventh Int. Joint Conf. on Autonomous Agents and Multiagent Systems*.

Cassandra, A.; Littman, M. L.; and Zhang, N. L. 1997. Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes. In *Proc. of the Thirteenth Conf. on Uncertainty in Artificial Intelligence*.

Dibangoye, J. S.; Mouaddib, A.; and Chaib-draa, B. 2009. Point-based incremental pruning heuristic for solving finite-horizon DEC-POMDPs. In *Proc. of the Eighth Int. Joint Conf. on Autonomous Agents and Multiagent Systems*.

Hansen, E. A.; Bernstein, D. S.; and Zilberstein, S. 2004. Dynamic programming for partially observable stochastic games. In *Proc. of the Nineteenth National Conf. on Artificial Intelligence*.

Nair, R.; Pynadath, D.; Yokoo, M.; Tambe, M.; and Marsella, S. 2003. Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. In *Proc. of the Nineteenth Int. Joint Conf. on Artificial Intelligence*.

Oliehoek, F. A.; Spaan, M. T. J.; and Vlassis, N. 2008. Optimal and approximate Q-value functions for decentralized POMDPs. *Journal of AI Research* 32.

Oliehoek, F. A.; Whiteson, S.; and Spaan, M. T. J. 2009. Lossless clustering of histories in decentralized POMDPs. In *Proc. of the Eighth Int. Joint Conf. on Autonomous Agents and Multiagent Systems*.

Seuken, S., and Zilberstein, S. 2007a. Improved memory-bounded dynamic programming for decentralized POMDPs. In *Proc. of the Twenty-Third Conf. on Uncertainty in Artificial Intelligence*.

Seuken, S., and Zilberstein, S. 2007b. Memory-bounded dynamic programming for DEC-POMDPs. In *Proc. of the Twentieth Int. Joint Conf. on Artificial Intelligence*.

Seuken, S., and Zilberstein, S. 2008. Formal models and algorithms for decentralized control of multiple agents. *Journal of Autonomous Agents and Multi-Agent Systems* 17(2).

Szer, D., and Charpillet, F. 2006. Point-based dynamic programming for DEC-POMDPs. In *Proc. of the Twenty-First National Conf. on Artificial Intelligence*.

Szer, D.; Charpillet, F.; and Zilberstein, S. 2005. MAA*: A heuristic search algorithm for solving decentralized POMDPs. In *Proc. of the Twenty-First Conf. on Uncertainty in Artificial Intelligence*.