Sparse Stochastic Finite-State Controllers for POMDPs

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

Abstract

Bounded policy iteration is an approach to solving infinite-horizon POMDPs that represents policies as stochastic finite-state controllers and iteratively improves a controller by adjusting the parameters of each node using linear programming. In the original algorithm, the size of the linear programs, and thus the complexity of policy improvement, depends on the number of parameters of each node, which grows with the size of the controller. But in practice, the number of parameters of a node with non-zero values is often very small, and it does not grow with the size of the controller. To exploit this, we develop a version of bounded policy iteration that manipulates a sparse representation of a stochastic finite-state controller. It improves a policy in the same way, and by the same amount, as the original algorithm, but with much better scalability.

Introduction

Partially observable Markov decision processes (POMDPs) provide a framework for decision-theoretic planning problems where actions need to be taken based on imperfect state information. Many researchers have shown that a policy for an infinite-horizon POMDP can be represented by a finite-state controller. In some cases, this is a deterministic controller in which a single action is associated with each node, and an observation results in a deterministic transition to a successor node (Kaelbling, Littman, and Cassandra 1998; Hansen 1998; Meuleau et al. 1999a). In other cases, it is a stochastic controller in which actions are selected based on a probability distribution associated with each node, and an observation results in a probabilistic transition to a successor node (Platzman 1981; Meuleau et al. 1999b; Baxter and Bartlett 2001; Poupart and Boutilier 2004; Amato, Bernstein, and Zilberstein 2007).

Bounded policy iteration (BPI) is an approach to solving infinite-horizon POMDPs that represents policies as stochastic finite-state controllers and iteratively improves a controller by adjusting the parameters of each node (which specify the action selection and node transition probabilities of the node) using linear programming (Poupart and Boutilier 2004). BPI is related to an exact policy iteration algorithm for POMDPs due to Hansen (1998), but provides an elegant and effective approach to approximation in which bounding the size of the controller allows a tradeoff between planning time and plan quality. Originally developed as an approach to solving single-agent POMDPs, BPI has also been generalized for use in solving decentralized POMDPs (Bernstein, Hansen, and Zilberstein 2005).

In BPI, the complexity of policy improvement depends on the size of the linear programs used to adjust the parameters of each node of the controller. In turn, this depends on the number of parameters of each node (as well as the size of the state space). In the original algorithm, each node of the controller has $|A| + |A||Z||N|$ parameters, where $|A|$ in the number of actions, $|Z|$ is the number of observations, and $|N|$ is the number of nodes of the controller. This assumes a fully-connected stochastic controller. In practice, however, most of these parameters have zero probabilities, and the number of parameters of a node with non-zero probabilities remains relatively constant as the number of nodes of the controller increases. Based on this observation, we propose a modified version of BPI that leverages a sparse representation of a stochastic finite-state controller. It improves the controller in the same way, and by the same amount, as the original algorithm. But it does so by solving much smaller linear programs, where the number of variables in each linear program depends on the number of parameters with non-zero values. Because the number of parameters of a node with non-zero values tends to remain relatively constant as the size of the controller grows, the complexity of each iteration of our modified version of BPI tends to grow only linearly with the number of nodes of a controller, which can be a dramatic improvement in scalability compared to the original algorithm.

Background

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

Variables: ϵ; $\psi_n(a), \forall a$; $\eta_n(a,z,n'), \forall a,z,n'$
Objective: Maximize ϵ
Improvement constraints:
$V_n(s) + \epsilon \leq \sum_a \psi_n(a) R(s,a) +$
$\gamma \sum_{a,z,n'} \eta_n(a,z,n') P(s'\|s,a) P(z\|s',a) V_{n'}(s'), \forall s$
Probability constraints:
$\sum_a \psi(a) = 1$
$\sum_{n'} \eta_n(a,z,n') = \psi(a), \forall a,z$
$\psi(a) \geq 0, \forall a$
$\eta_n(a,z,n') \geq 0, \forall a,z,n'$

Table 1: Linear program for improving a node n of a stochastic finite-state controller.

Since the state of the environment cannot be directly observed, we let b denote an $|\mathcal{S}|$-dimensional vector of state probabilities, called a *belief state*, where $b(s)$ denotes the probability that the system is in state s. If action a is taken and followed by observation z, the successor belief state, denoted b^a_z, is determined using Bayes' rule.

Policy representation and evaluation

A *policy* for a POMDP can be represented by a finite-state controller (FSC). A stochastic finite-state controller is a tuple $< \mathcal{N}, \psi, \eta >$ where $\mathcal{N}$ is a finite set of nodes, ψ is an action selection function that specifies the probability, $\psi_n(a) = Pr(a|n)$, of selecting action $a \in A$ when the FSC is in node $n \in \mathcal{N}$, and η is a node transition function that specifies the probability, $\eta_n(a,z,n') = Pr(n'|n,a,z)$, that the FSC will make a transition from node $n \in \mathcal{N}$ to node $n' \in \mathcal{N}$ after taking action $a \in A$ and receiving $z \in Z$.

The value function of a policy represented by a FSC is piecewise linear and convex, and can be computed exactly by solving the following system of linear equations, with one equation for each pair of node $n \in \mathcal{N}$ and state $s \in S$:

$$V_n(s) = \sum_{a \in A} \psi_n(a) R(s,a) + \beta \sum_{a,z,s,n'} \eta_n(a,z,n') P(s'|s,a) P(z|s',a) V_{n'}(s'). \quad (1)$$

In this representation of the value function, there is one $|S|$-dimensional vector V_n for each node $n \in \mathcal{N}$ of the controller. The value of any belief state b is determined as follows,

$$V(b) = \max_{n \in N} \sum_{s \in S} b(s) V_n(s), \quad (2)$$

and the controller is assumed to start in the node that maximizes the value of the initial belief state. The value function of an optimal policy satisfies the following optimality equation,

$$V(b) = \max_{a \in A} \left\{ R(b,a) + \beta \sum_{z \in Z} P(z|b,a) V(b^a_z) \right\}, \quad (3)$$

where $R(b,a) = \sum_{s \in S} b(s) R(s,a)$ and $P(z|b,a) = \sum_{s \in S} b(s) \sum_{s' \in S} P(s'|s,a) P(z|s',a)$.

Algorithm 1 Bounded policy iteration

```
repeat
  repeat
    Solve the linear system given by Equation (2)
    for each node n of the controller do
      solve the linear program in Table 1
      if ε > 0 then
        update parameters and value vector of node
      end if
    end for
  until no improvement of controller
  find belief states reachable from tangent points
  create k new nodes that improve their value
until no new node is created
```

Bounded policy iteration

Policy iteration algorithms iteratively improve a policy by alternating between two steps: policy evaluation and policy improvement. Hansen (1998) proposed a policy iteration algorithm for POMDPs that represents a policy as a deterministic finite-state controller. In the policy improvement step, it uses the dynamic programming update for POMDPs to add, merge and prune nodes of the controller. The algorithm is guaranteed to converge to an ϵ-optimal controller and can detect convergence to an optimal policy.

A potential problem is that the number of nodes added in the policy improvement step is often very large, and the controller can grow substantially in size from one iteration to the next. Because the complexity of the policy improvement step increases with the size of the controller, allowing the size of the controller to grow too fast can slow improvement and limit scalability.

To address this problem, Poupart and Boutilier (2004) proposed a *bounded policy iteration* algorithm that controls growth in the size of the controller in two ways. First, a policy is represented as a stochastic finite-state controller that can be improved without increasing its size, by adjusting the action and node-transition probabilities of each node of the controller using linear programming. Second, when the controller cannot be improved further in this way, k nodes are added to the controller, where k is some small number greater than or equal to one. We review each of these two steps in further detail.

Improving nodes The first step attempts to improve a controller while keeping its size fixed. For each node n of the controller, it solves the linear program in Table 1. The linear program searches for action probabilities and node transition probabilities for the node that will improve the value vector V_n associated with the node by some amount ϵ for each state, where ϵ is the objective maximized by the linear program. If an improvement is found, the parameters of the node are updated accordingly. (The value vector may also be updated, and will be further improved during the policy evaluation step.)

Poupart and Boutilier (2004) show that the linear program can be interpreted as follows: it implicitly considers the vectors of the backed-up value function that would be created

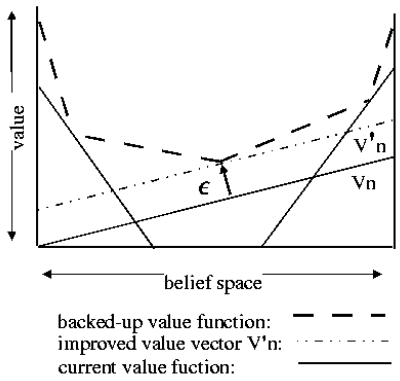

Figure 1: BPI can improve the value vector V_n by an amount ϵ to obtain the improved value vector V_n', which is tangent to the backed-up value function.

by performing a dynamic programming update. In particular, the linear program searches for a convex combination of these backed-up vectors that pointwise dominates the value vector currently associated with the node. If an improvement is found, the parameters of the convex combination become the new parameters of the node. This interpretation is illustrated in Figure 1, which is adapted from a similar figure from Poupart's dissertation (2005). As Figure 1 shows, the new value vector is parallel to the old vector (i.e., the value of each component is improved by same amount) and it is tangent to the backed-up value function.

Adding nodes Eventually, no node can be improved further by solving its corresponding linear program. At this point, BPI is at a local optimum. It can escape such a local optimum by adding one or more nodes to the controller.

The key insight is that when a local optimum is reached, the value vector of each node of the controller is tangent to the backed-up value function at one or more belief states. Moreover, the solution of the linear program that is the dual of the linear program in Table 1 is a belief state that is tangent to the backed-up value function; it is called the *tangent belief state*. (Since most linear program solvers return the dual solution whenever they solve a linear program, we assume that we get the tangent belief state when we solve the linear program in Table 1, in addition to the value of ϵ and the action and node transition probabilities.)

To escape a local optimum, it is necessary to improve the value of the tangent belief state. This leads to the method for adding nodes to a controller. Given a tangent belief state b, the algorithm considers every belief state b' that can be reached from it in one step (i.e., by taking some action a followed by some observation z). For each reachable belief state, a backup is performed (as defined by Equation 3). If the backed-up value is better than the value of the belief state based on the current value function, a deterministic node is added to the controller that has the same action, and, for each observation, the same successor node, that created the improved backed-up value. Usually, it is only necessary to add a single node to the controller to escape a local optimum. Because a value vector may be tangent to the backed-up value function for a linear portion of belief space, however, it may be necessary to add more than one node to escape the local optimum. As we will see, this method for adding nodes to escape local optima is related to the method for exploiting sparsity that we develop in this paper.

Two sources of complexity Most of the computation time of BPI is spent in solving the linear programs that are used to adjust the parameters that specify the action selection probabilities and node transition probabilities of each node of a controller. The size of the linear programs, and thus the complexity of BPI, depends on two things: the size of the controller (which determines the number of variables in the linear program) and the size of the state space (which determines the number of constraints).

In this paper, we focus on the first source of complexity, that is, we focus on improving the complexity of BPI with respect to controller size. Coping with POMDPs with large state spaces is an orthogonal research issue, and several approaches have been developed that can be combined with BPI. For example, Poupart and Boutilier (2005) describe a technique called *value-directed compression* and report that it allows BPI to solve POMDPs with up to 33 million states. Because the test problems used later in this paper have small state spaces, it is important to keep in mind that the techniques developed in the next section for improving the scalability of BPI with respect to controller size can be combined with techniques for coping with large state spaces.

Sparse bounded policy iteration

In this section, we describe a modified version of BPI that we call *Sparse BPI*. In each iteration, it improves a FSC by the same amount as the original algorithm, but with much improved scalability. To motivate our approach, we begin with a discussion of the sparse structure of stochastic finite-state controllers found by BPI.

Sparse stochastic finite-state controllers

As we have seen, each iteration of BPI solves $|\mathcal{N}|$ linear programs, and each linear program has $|A| + |A||Z||\mathcal{N}|$ variables and $|S| + |A||Z|$ constraints (in addition to the constraints that the variables have non-negative values). Even for small FSCs, the number of variables in the linear programs can be very large, and the fact that the number of variables grows with the number of nodes in the controller significantly limits the scalability of BPI.

If we look at the controllers produced by BPI, however, we find that most of the parameters of each node (i.e., most of the variables in the solutions of the linear program) have values of zero. Table 2 illustrates this vividly for two benchmark POMDPs from (Cassandra 2004). The overwhelming majority of each node's parameters have zero probabilities. This is despite the fact that *all* of the nodes of the controllers are stochastic. For the Slotted Aloha problem, a deterministic node has 4 parameters, one for a choice of action, and 3 to specify the successor node for each of the 3 observations. In Table 2, the minimum number of non-zero parame-

Test problem	Statistic	Number of nodes of controller 50	100	150	200	250	300
Slotted Aloha $\|S\| = 30, \|A\| = 9, \|Z\| = 3$	total parameters per node	1,359	2,709	4,059	5,409	6,759	8,109
	min non-zero parameters	6	5	7	7	7	6
	avg non-zero parameters	11	11	11	11	11	11
	max non-zero parameters	17	18	20	21	21	21
Hallway $\|S\| = 60, \|A\| = 5, \|Z\| = 21$	total parameters per node	5,255	10,505	15,755	21,005	26,255	31,505
	min non-zero parameters	28	44	32	35	36	35
	avg non-zero parameters	99	112	105	103	102	100
	max non-zero parameters	130	151	158	157	158	158

Table 2: Total number of parameters per node of stochastic finite-state controllers found by bounded policy iteration, and minimum, average, and maximum number of parameters with non-zero values, as a function of the size of the controller. (The average is rounded up to the nearest integer.) The two test POMDPs are from (Cassandra 2004).

ters for any node is always greater than this, which indicates that all of the controller nodes are stochastic. For the hallway problem, a deterministic node has 22 parameters, one for a choice of action, and 21 to specify the successor node for each of the 21 observations. Again, the minimum number of non-zero parameters for any node is greater than this. For these problems and many others, BPI is very effective in leveraging the possibility of stochastic nodes to improve a controller without increasing its size, but the actual number of parameters with non-zero probabilities is a very small fraction of the total number of parameters.

Besides the sparsity of the stochastic finite-state controllers found by BPI, an equally important observation about the results in Table 2 is that the number of parameters with non-zero probabilities tends to remain the same as the controller grows in size, whereas the total number of parameters grows significantly. In the following, we develop a modified version of BPI that exploits this sparse structure.

Sparse policy improvement algorithm

We begin by describing the main idea of the algorithm. We have seen that the number of parameters of a node with non-zero probabilities in the solution of the linear program of Table 1 is very small relative to the total number of parameters. Let us call these the *useful parameters*. If we could somehow identify the useful parameters of a node, we could improve a node by solving a linear program that only includes these variables. We call a linear program that only includes some of the parameters of a node a *sparse linear program*. Our approach will be to solve a series of sparse linear programs, where the last sparse linear program is guaranteed to include all of the useful parameters of the node. This approach will get the same result as solving the full linear program of Table 1, and can be much more efficient if the sparse linear programs are much smaller.

We next describe an iterative method for identifying a small set of parameters that includes all of the useful parameters. Our method starts with the parameters of the node that currently have non-zero probabilities. This guarantees a solution that is at least as good as the current node (and possibly better). Then we add parameters using a technique that is inspired by the technique that BPI uses to add nodes to a controller in order to escape local optima. As shown in Figure 1, the value vector that is created by solving the linear program in Table 1 is tangent to the backed-up value function, and the solution of the dual linear program is a tangent belief state. In the case of a sparse linear program, however, we have a *partial backed-up value function* that does not include vectors corresponding to parameters that are not included in the sparse linear program. (For example, if only one action parameter is included in the sparse linear program, the partial backed-up value function does not include any vectors created by taking a different action.)

If the sparse linear program is missing some useful parameters, it must be possible to improve the value of the tangent belief state by considering the vectors of the full backed-up value function. So, to identify potentially useful parameters, we perform a backup for the tangent belief state. If the backed-up value is better than the value of the tangent belief state based on the current value vector, we add to our set of parameters the parameters used to create the backed-up value: the best action, and, for each observation, the best successor node. Then we solve another sparse linear program that includes these parameters in addition to the parameters contained in the previous sparse linear program.

If the value vector is tangent to the partial backed-up value function at a single belief state, adding the parameters that improve the backed-up value of this tangent belief state improves the value vector for this node. But since the value vector may be tangent to the partial backed-up value function along a linear segment, adding parameters does not guarantee improvement of the node. In this case, however, it does change the result of the new sparse linear program in an important way. Because adding these parameters to the sparse linear program improves the partial backed-up value function that is searched by the linear program, the value vector is no longer tangent to the partial backed-up value function at the same belief state. In other words, even if the solution of the sparse linear program is not changed by adding these parameters, the solution of the dual linear program changes in one important way: there must be a new tangent belief state. This is illustrated in Figure 2.

This points to an iterative algorithm for improving a node. At each step, the algorithm solves a sparse linear program, and then performs a backup for the tangent belief state. If the backed-up value of the tangent belief state is better than

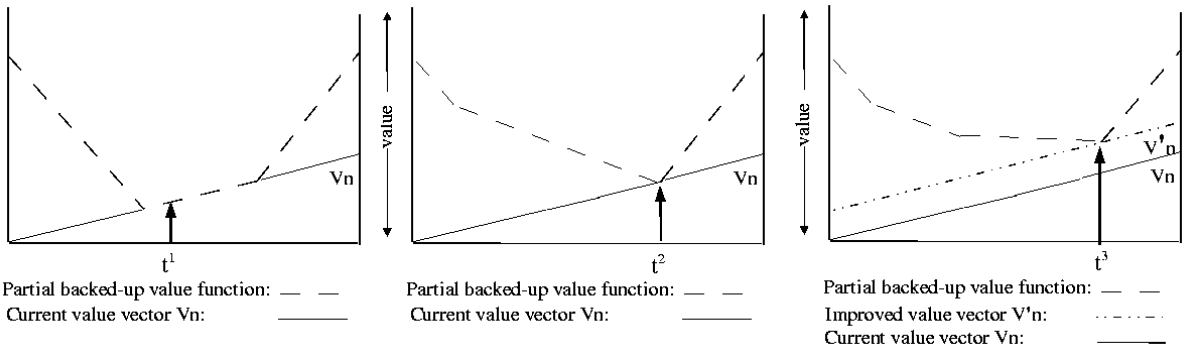

Figure 2: In the left panel, the value vector is tangent to the partial backed-up value function along a linear segment. The center panel shows the result of adding parameters to the sparse linear program that improve the partial backed-up value function at tangent belief state t^1. Although this does not improve the value vector, it creates a new tangent belief state t^2. When parameters are added to the sparse linear program that improve its backed-up value, the result is an improved value vector with a new tangent belief state t^3, as shown in the right panel.

its value based on the current value vector, the parameters that produced the improved backed-up value are added to the sparse linear program. Since the backed-up value of the tangent belief state is improved, it must be the case that at least some of these parameters are not in the current sparse linear program; if they were, the current value vector would not be tangent to the partial backed-up value function at this belief state. The condition for terminating this iterative process of adding parameters to a sparse linear program is based on the following lemma.

Lemma 1 *The difference between the backed-up value of a tangent belief state and its value based on the current value vector bounds the amount by which the linear program in Table 1 can improve the value vector of a node.*

In other words, if the backed-up value of the tangent belief state is the same as its value based on the current value vector, it is impossible to improve the value vector further. This follows from the fact that possible improvement of the value any belief state by solving the linear program is bounded by the backed-up value of that belief state.

This iterative algorithm is guaranteed to terminate because (i) whenever the backed-up value of the tangent belief state is greater than its value based on the value vector created by the current sparse linear program, at least one (and no more than $(1+|Z|)$ parameters will be added to the sparse linear program, and (ii) the total number of parameters is finite. Moreover, the last sparse linear program solved in this iterative process will have the same result as solving the full linear program in Table 1. This follows from the fact that the process terminates when the difference between the backed-up value of the tangent belief state (for the last sparse linear program) and the value of the tangent belief state based on the current value vector of the node is zero, which from Lemma 1 means that no further improvement is possible.

Theorem 1 *This method of improving a stochastic node by solving a sequence of sparse linear program is guaranteed to terminate and the last sparse linear program produces the same result as solving the full linear program in Table 1.*

(Of course, to say that the last sparse linear program produces the same result means that it produces the same value vector. There may be different parameter settings that produce the same value vector.)

Because the parameters added to the initial sparse linear program are useful parameters, each of the sparse linear programs solved in this iterative process is typically very small compared to the full linear program of Table 1. (This is the case whenever that the FSCs found by BPI are very sparse.) We call this iterative approach to improving the nodes of a stochastic FSC *sparse policy improvement*. The high-level pseudocode is shown in Algorithm 2.

It should be clear that this approach is somewhat related to the way the original BPI algorithm adds nodes to a controller. In both cases, the technique for breaking a local optimum at a tangent belief state is to improve the backed-up value function of the tangent belief state. In the original BPI algorithm, the value of the tangent belief state is improved by adding nodes to the controller. In sparse policy improvement, it is improved by adding parameters to a node.

Algorithm 2 Sparse policy improvement

```
for each node of controller do
  Create initial sparse linear program
  {its parameters are the parameters of the node that currently have non-zero probabilities}
  repeat
    Solve sparse linear program for the node
    if ε > 0 then
      Update parameters and value vector of the node
    end if
    Do backup for tangent belief state
    if backed-up value is improvement then
      Add parameters that produced backed-up value
    end if
  until backup does not improve value of tangent belief
end for
```

Test problem	Algorithm	Number of nodes of controller 50	100	150	200	250	300
Slotted Aloha	BPI	202	415	684	871	1,085	1,620
$\|S\| = 30, \|A\| = 9, \|Z\| = 3$	Sparse-BPI	16	19	19	20	21	23
Hallway	BPI	3,900	10,180	17,340	24,485	27,428	32,973
$\|S\| = 60, \|A\| = 5, \|Z\| = 21$	Sparse-BPI	1,215	935	1,110	973	1,094	1,267

Table 3: Average time (in CPU milliseconds) for improving a single node of a finite-state controller, as a function of the size of the controller, for two benchmark POMDPs.

Experimental results

We implemented sparse bounded policy iteration and tested it successfully on several benchmark POMDPs. Table 3 shows our results for the two POMDPs considered earlier in Table 2. The experiments were run on a 3.0 GHz processor, using CPLEX version 9.0 as the linear program solver.

Table 3 shows that Sparse BPI is much more efficient that the original BPI algorithm in improving sparse stochastic FSCs. Even for relatively small controllers of 300 nodes, Sparse BPI is between 30 and 80 times faster than BPI. More importantly, its relative advantage grows with the size of the controller. Because the size of the linear programs solved by Sparse BPI remains about the same as the controller grows in size, in contrast to BPI, the complexity of an iteration of Sparse BPI tends to grow only linearly with the size of the controller (at least for FSCs with sparse structure).

(There may be a slight increase in the per-node running time of Sparse BPI as the size of the controller grows, since the complexity of backups grows slightly when there are more value vectors to search in evaluating a belief state. But if the average number of non-zero parameters per node does not grow with the size of the controller, the per-node running time for solving the linear programs will not grow either.)

An interesting difference between Sparse BPI and BPI is that the running time of an iteration of Sparse BPI depends on how much improvement of the controller is possible. If all nodes of a controller are already at a local optimum, Sparse BPI often needs to solve only one or two linear programs per node in order to determine that further improvement is not possible. In this case, an iteration of Sparse BPI terminates relatively quickly. But if much improvement of the FSC is possible, Sparse BPI often needs to solve ten to twenty sparse linear programs for some nodes in order to add all of the parameters that are needed to maximize improvement of the node. So far, the largest difference we have seen in the running time of Sparse BPI as a function of how much improvement of the FSC is possible is about a factor of four. To obtain reliable running times for Sparse-BPI, the results in Table 3 are averaged over several iterations of Sparse-BPI.

This observation about the behavior of Sparse BPI suggests that a simple variation of the algorithm could offer a tradeoff between improvement and running time. Instead of continuing to add parameters until the difference between the backed-up value of the tangent belief state and its current value is zero, Sparse-BPI could stop adding parameters as soon as the difference is small enough to demonstrate that only a small amount of further improvement is possible.

Conclusion

We have presented a modified bounded policy iteration algorithm for POMDPs called sparse bounded policy iteration. The new algorithm exploits the sparse structure of stochastic finite-state controllers found by bounded policy iteration. Each iteration of the algorithm produces the identical improvement of a controller that an iteration of the original bounded policy iteration algorithm produces, but with much improved scalability. Whereas the time it takes for the original algorithm to improve a single node grows with size of the controller, the time it takes for the new algorithm to improve a single node is typically independent of the size of the controller. This makes it practical to use bounded policy iteration to find larger controllers for POMDPs.

References

Amato, C.; Bernstein, D.; and Zilberstein, S. 2007. Solving POMDPs using quadratically constrained linear programs. In *Proceedings of IJCAI-07*, 2418–2424.

Baxter, J., and Bartlett, P. 2001. Infinite-horizon policy-gradient estimation. *JAIR* 15:319–350.

Bernstein, D.; Hansen, E.; and Zilberstein, S. 2005. Bounded policy iteration for decentralized POMDPs. In *Proceedings of IJCAI-05*, 1287–1292.

Cassandra, A. 2004. Tony's POMDP file repository page. http://pomdp.org/pomdp/examples/index.shtml.

Hansen, E. 1998. Solving POMDPs by searching in policy space. In *Proceedings of UAI-98*, 211–219.

Kaelbling, L.; Littman, M.; and Cassandra, A. 1998. Planning and acting in partially observable stochastic domains. *Artificial Intelligence* 101:99–134.

Meuleau, N.; Kim, K.; Kaelbling, L.; and Cassandra, A. 1999a. Solving POMDPs by searching the space of finite policies. In *Proceedings of UAI-99*, 417–426.

Meuleau, N.; Peshkin, L.; Kim, K.; and Kaelbling, L. 1999b. Learning finite-state controllers for partially observable environments. In *Proceedings of UAI-99*, 427–436.

Platzman, L. 1981. A feasible computational approach to infinite-horizon partially-observed Markov decision problems. Technical report, Georgia Tech. Reprinted in Working Notes of the AAAI Fall Symposium on Planning using POMDPs, 1998.

Poupart, P., and Boutilier, C. 2004. Bounded finite state controllers. In *NIPS 16*, 823–830.

Poupart, P., and Boutilier, C. 2005. VDCBPI: An approximate scalable algorithm for large POMDPs. In *NIPS 17*.

Poupart, P. 2005. *Exploiting Structure to Efficiently Solve Large Scale Partially Observable Markov Decision Processes*. Ph.D. Dissertation, University of Toronto.