Hierarchy and Finite-State Controllers for POMDPs

Daniel S. Bernstein
June 27, 2002

1 Introduction

Many complex POMDPs have optimal or near-optimal policies that can be represented by small finite-state controllers. This fact has motivated researchers to develop POMDP planning algorithms that search in the space of such controllers (Hansen, 1998; Meuleau, Kim, Kaelbling & Cassandra, 1999).

These algorithms can solve some small problems efficiently, but they take far too long to find good policies for more complex problems. One common way of speeding up planning is to incorporate prior knowledge in the form of a pre-specified macro-action hierarchy. For example, it has been shown that options can accelerate planning in MDPs, as long as the options are chosen intelligently (Sutton, Precup & Singh, 1999). Can we scale up the aforementioned POMDP planning algorithms in a similar way?

In this report I describe preliminary worked aimed at answering this question. First, I formally define observation-based options (or obtions) for POMDPs. Next, I extend Hansen's policy iteration algorithm to work with obtions. Finally, I present experimental results with different obtion sets in a simple domain. Although the results are not conclusive, I learned some important lessons from my experiments. I saw that planning with obtions can be faster than planning with just primitive actions. However, using primitive actions to *refine* an obtion-based policy can be difficult. I also learned that obtions which are useful regardless of the initial state distribution are probably the best kind to use with Hansen's policy iteration.

There has been recent related work on this problem. Hernandez-Gardiol and Mahadevan (2000) supply knowledge in the form of a partial policy and learn the missing pieces using the Nearest Sequence Memory algorithm (McCallum, 1995). Theocharous and Mahadevan (2002) extend the hierarchical hidden Markov model to allow for actions and rewards, and present heuristics for solving the resulting problem. The main difference between

my work and the work I just mentioned is that I use finite-state controllers at each level of the hierarchy. Since each level is a finite-state controller, the entire hierarchy can be "unrolled" into one large finite-state controller. This homogeneity helps us smoothly integrate hierarchy into existing algorithms and maintain formal convergence guarantees.

2 POMDPs and Obtions

Recall that a POMDP is a tuple $\langle S, A, P, R, \Omega, O, \Pi \rangle$, where S is a state set; A is an action set; P is a table of transition probabilities; R is a reward function; Ω is an observation set; O is a table of observation probabilities; and Π is an initial state distribution.

I define an *observation-based option (obtion)* for a POMDP to be a pair $\langle C, \pi \rangle$. The *continuation set*, $C \subseteq \Omega^*$, is the set of observation sequences for which the obtion policy is defined. We require that $\epsilon \in C$, where ϵ is the empty sequence. The *policy* $\pi : C \to A$ assigns an action to each observation sequence in the continuation set. The obtion chooses actions according to π and terminates when the sequence of observations since initiation falls outside of C. Note that a primitive action is just an obtion with $C = \{\epsilon\}$.

Suppose we are given a POMDP and an obtion set B. A *policy over obtions* is a mapping $\gamma : \Omega^* \to B$. The *value* of a policy γ is the expected discounted reward from executing γ from the initial state distribution. An optimal policy over obtions is simply one with maximal value.

It is important to note the differences between obtions and options for MDPs (Sutton, Precup & Singh, 1999). In an MDP, the first action taken by an option can depend on the state from which the option was initiated, and the option can terminate upon a transition to a certain state. The obtions that I have defined for POMDPs are "shifted" in time. They both start and end with an action being taken. Thus it is not possible to design an obtion that tells a robot to move forward until it observes a wall. It must take one more action after it reaches the wall. Also, since obtions begin with an action, there is no way to restrict the set of states from which the obtion can be executed. The main reason why I define obtions this way is that it turns out to be a useful definition for most POMDP planning algorithms. For future work, it would be nice to devise a more natural definition that still works with existing algorithms.

I should also note that one could define obtions that call other obtions instead of just primitive actions. I do not formally define obtions this way, but I believe it is straightforward to extend all of the results in this report

to these more general obtions.

3 Obtions as Finite-State Controllers

One natural way to express obtions is with finite-state controllers. A *finite-state controller (FSC)* is a tuple $\langle M, \alpha, \delta, T \rangle$, where M is a set of controller states, with m_0 a distinguished initial state; $\alpha : M \to A$ is a mapping from controller states to actions; $\delta : M \times \Omega \to M$ is a transition function; and $T \subseteq M$ is a set of terminal states. The controller takes as input a sequence of observations and makes state transitions accordingly. The policy for the corresponding obtion is determined by the actions taken from each of the controller states. The observation sequences contained in the continuation set are exactly those with no prefix leading the finite-state controller into a terminal state.

For this paper, I restrict attention to obtions that are represented as FSCs. It should be noted, however, that there are obtions which cannot be represented with any finite-state controller. For instance, suppose we have a POMDP with action set $A = \{a_0, a_1\}$ and observation set $\Omega = \{o_0, o_1\}$. One obtion for this POMDP that can't be represented as an FSC has $C = \Omega^*$, and $\pi(\overline{o}) = a_0$ if and only if $\overline{o}$ has an equal number of o_0's and o_1's. Intuitively, this cannot be represented with an FSC because it requires an unbounded amount of memory.

4 Hansen's Policy Iteration

In this section I will briefly summarize Hansen's policy iteration algorithm for planning with primitive actions in infinite-horizon POMDPs (which I will hereafter refer to as HPI). A more detailed description can be found in (Hansen, 1998). The algorithm represents policies as FSCs with no special initial states or terminal states. It starts with an arbitrary FSC, and each iteration produces a new FSC. The algorithm terminates when the Bellman residual for the current FSC falls below a small value.

An iteration consists of three phases: policy evaluation, policy improvement, and pruning. First, the current controller is evaluated, yielding a piecewise linear value function with a vector for each machine state. Next, a dynamic programming update (in my implementation, incremental pruning) is used to back up the value over one step, and a new piecewise linear value function is created. For each new vector that is added in the update, a new state is added to the machine. Finally, some of the useless controller

states are pruned away. One nice property is that it is possible for the FSC to shrink as well as grow over the course of the algorithm.

When executing an FSC policy in a POMDP, the initial machine state for the controller is set to be the one that has the highest value with respect to the start state distribution of the POMDP.

5 Policy Iteration with Obtions

5.1 Planning with Just Obtions

The way I have defined obtions makes it relatively easy to extend HPI to work with them. Each obtion is an FSC, so a policy over obtions can still be represented as an FSC. Policy evaluation is similar to the original algorithm, except now vectors are produced only for the initial states of the obtions in the policy. The internal obtion states are in a sense not "accessible", so their values do not need to be computed. Pruning remains the same with obtions, except now whole obtions are pruned at once.

The dynamic programming update requires a bit of explanation. The best way to understand the difference between the primitive action case and the obtion case is to look at the worst case run times for each. In the primitive action case, suppose the current FSC has n states. We must consider all possible ways of taking an action and transitioning to a controller state. For each action there are $n^{|\Omega|}$ ways of transitioning. Thus the DP update takes $|A|n^{|\Omega|}$ steps. In the obtion case, suppose the current FSC has n obtions (and thus n different initial obtion states). We must consider all possible ways of executing an obtion and transitioning to the initial state of an obtion in the FSC. Recall that T is the set of termination states for an obtion. For each obtion, there are $n^{|T||\Omega|}$ possible ways of transitioning. Now the DP update is exponential in another parameter: the number of terminal states for the obtions.

I believe that with the appropriate modifications, HPI is guaranteed to converge to the optimal policy with respect to whatever obtion set it is given. I leave the formal proof of this to future work.

5.2 Refining an Obtion-Based Policy

I have argued that, in theory, it is not difficult to extend HPI so that it converges to the optimal policy with respect to any given obtion set. However, what if we have an optimal policy for an obtion set, and we want to reintroduce primitive actions to get convergence to a truly optimal policy? If we

Figure 1: A simple POMDP.

can do this efficiently, then we will have a well-behaved *anytime* algorithm. It turns out that we can smoothly transition from obtions to primitive actions in HPI. At any point, we can take the current controller involving obtions, and view each obtion not as indivisible but as a mini-FSC that can be adapted. We then have an FSC over FSCs, which is just a big FSC over primitive actions. This controller can be used as the starting point for HPI with primitive actions, and we are guaranteed convergence to optimality in the limit.

6 Experiments

6.1 A Simple POMDP

For experimentation, I used a simple seven-state gridworld POMDP (see Figure 1). The agent starts at either the far left or the far right, each with equal probability. The goal state is located in the middle. Upon entry into the goal state, the agent receives a reward of 1, and rewards of zero from then on. The discount rate is set to 0.75 to encourage the agent to reach the goal state as quickly as possible. The agent has four actions: move left (ML), move right (MR), observe left (OL), and observe right (OR). If the agent executes ML or MR and does not have a wall in its way, it deterministically moves one step in the corresponding direction, and observes nothing. If there is a wall in the way, the agent remains in its previous state and observes nothing. When the agent executes OL, it is told whether or not it has a wall directly to the left of it, and similarly for OR.

One optimal policy contains OL as the first action, and then MR three times in a row or ML three times in a row, depending on what was observed. The controller for this policy is shown in Figure 2. Its value is 0.421875.

6.2 Primitive Actions

My first experiment consisted of running HPI on the problem, using only primitive actions. This took 7 iterations to converge. Some of the controllers produced are shown in Figure 3. Table 1 shows the size of the controller, the value of the controller, and the total time after each iteration. (It is

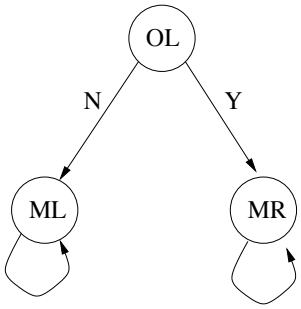

Figure 2: An optimal controller for the POMDP.

Table 1: Comparison of the different obtion sets

Iter.	**Primitive**			**Useful**			**Better**		
	size	value	time	size	value	time	size	value	time
0	1	0.281	0.00	1	0.281	0.00	$1 \rightarrow 3$	0.421	0.00
1	2	0.281	0.00	2	0.399	0.00	7	0.421	0.01
2	3	0.281	0.00	2	0.399	0.00	14	0.421	0.02
3	6	0.399	0.00	$2 \rightarrow 6$	0.399	0.00	13	0.421	0.06
4	10	0.421	0.01	12	0.421	0.01	11	0.421	0.08
5	13	0.421	0.04	17	0.421	0.04			
6	12	0.421	0.06	11	0.421	0.07			
7	10	0.421	0.09	10	0.421	0.09			

questionable how much information is contained in the total time, as this depends heavily on implementation details, which I didn't pay much attention to for this project.)

6.3 A Useful Obtion Set

Next, I created a set of two obtions that seemed like they would be useful for this problem. The obtions are to move left three times in a row (GL) and move right three times in a row (GR). The optimal policy for this option set was found after three iterations. It is shown in Figure 4, along with the equivalent controller involving just primitive actions. I used this controller as the starting point of HPI with primitive actions. This converged after four more iterations. The time and controller size data are contained in the table.

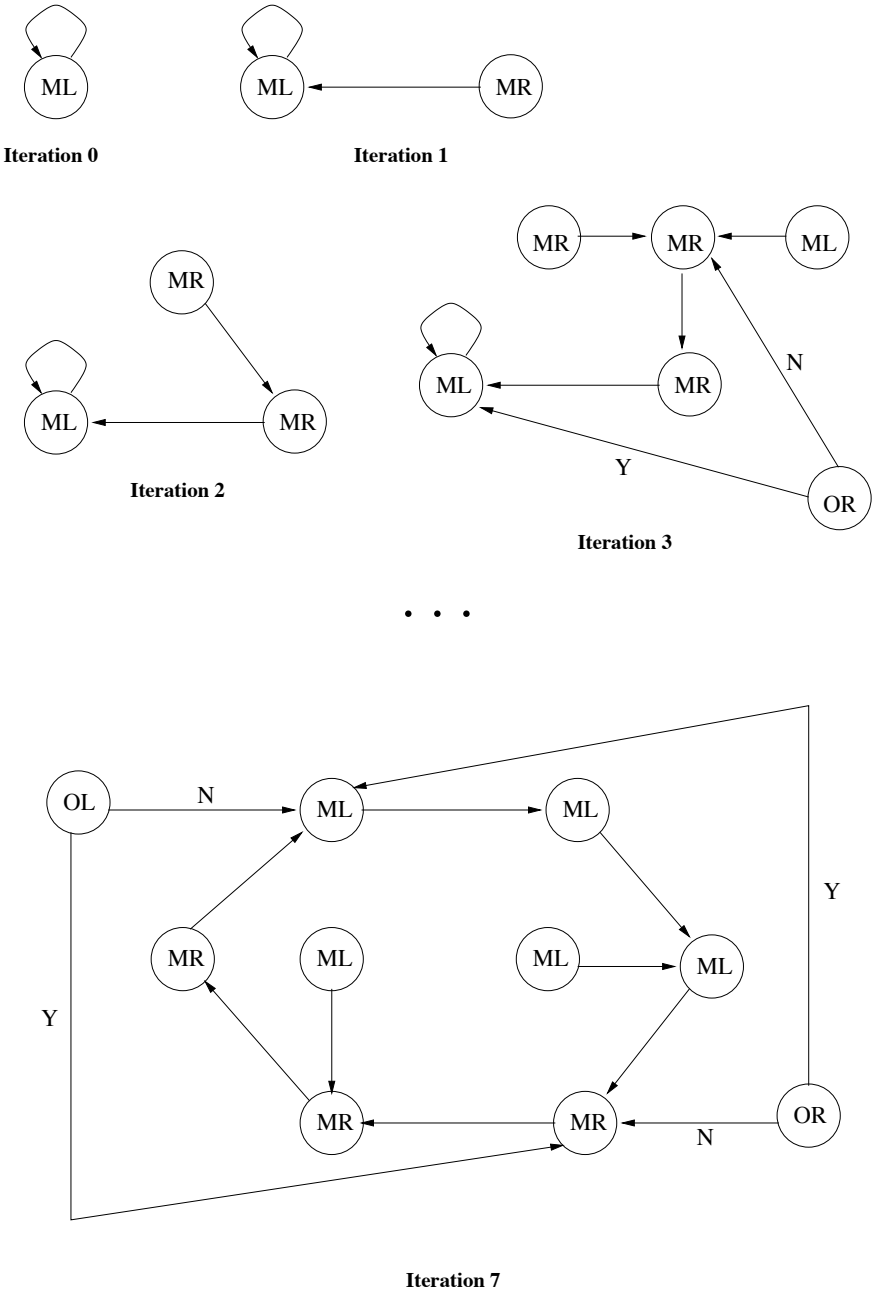

Figure 3: The sequence of controllers produced by HPI.

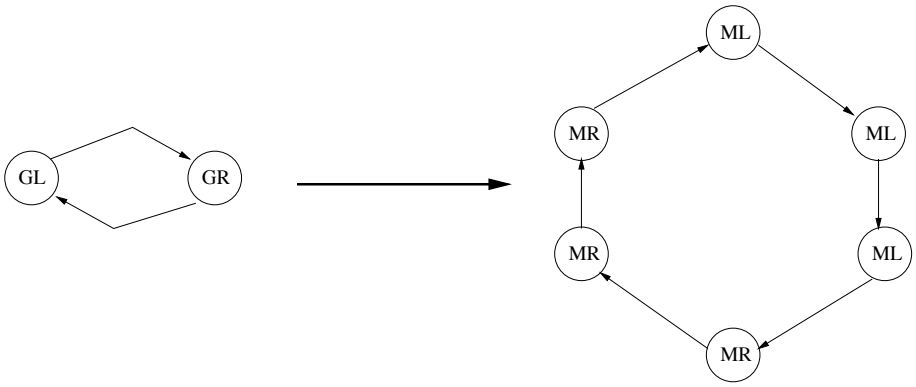

Figure 4: The optimal policy for this option set.

As expected, we get faster convergence to optimality w.r.t. the obtion set. However, it is interesting that in the process of refining this controller, we produce one with 17 states – more than any produced in the previous experiment.

6.4 A Better Obtion Set

I decided to repeat the previous experiment with an obtion set containing only one obtion – the optimal policy described earlier. One might expect that with this controller as a starting point, HPI would converge after just one iteration. Somewhat surprisingly, the algorithm actually requires four iterations to converge (see the table for details). This result becomes less surprising when we analyze HPI more carefully. It turns out that HPI completely ignores the initial state distribution of the POMDP. It does not converge until it has found a controller that is optimal regardless of the initial state distribution.

Suppose that instead of the initial state distribution I used, I had one in which the initial state set included the states directly adjacent to the end states. Neither of these states is next to a wall, so observing on the first step is suboptimal. The optimal policy actually involves going out to the wall and then moving back towards the center. Thus the obtion I started with was too specific to the initial state distribution.

7 Conclusions and Future Work

I described an approach to extending Hansen's policy iteration for POMDPs to work with obtions. It is possible to do this and see benefits, but the benefits depend largely on the particular set of obtions used. I learned two important lessons from my experiments. First, refining an obtion-based controller to get the truly optimal policy can result in large controllers being formed. Second, obtions whose utility is strongly dependent on the start state distribution are not the best type to use with policy iteration.

I don't have a solution to the first problem, and there may be no elegant solution. But for the second, it may be possible to integrate obtions into POMDP algorithms that don't ignore the initial state distribution. An example of such an algorithm is Hansen's heuristic search for POMDPs (Hansen, 1998). This algorithm replaces the policy improvement part of policy iteration with a search that looks forward from the actual initial state distribution.

Another avenue for future work is to study the literature on automata minimization to see whether there are ideas that apply to solving POMDPs. If a finite-state controller possesses symmetry, then there may exist a controller with less states that produces the exact same behavior. Despite the fact that Hansen's algorithm prunes states at each iteration, it is unknown whether the reduced automaton is guaranteed to be minimal. (Hansen believes that this guarantee exists, but I have yet to see a convincing argument.) Automata minimization ideas seem to become more relevant when we consider POMDPs with options. Sometimes there may be no symmetry in a controller unless it is looked at from a high level perspective. Perhaps there are ways to construct obtions to increase the likelihood that controllers using them possess symmetry.

Finally, in order to really understand the behavior of policy iteration with obtions, we need to run experiments in more complex domains. A major challenge is to choose domains and obtion sets in such a way that the controllers produced by policy iteration don't get too large too quickly.

References

Hansen, E. (1998). Solving POMDPs by searching in policy space. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence* (pp. 211–219).

Hernandez-Gardiol, N. & Mahadevan, S. (2000). Hierarchical memory-based

reinforcement learning. In *Proceedings of Advances in Neural Information Processing Systems 15*.

McCallum, A. K. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, Rochester, NY.

Meuleau, N., Kim, K.-E., Kaelbling, L. & Cassandra, A. R. (1999). Solving POMDPs by searching the space of finite policies. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence* (pp. 417–426).

Sutton, R. S., Precup, D. & Singh, S. (1999). Between MDPs and Semi-MDPs: Learning, planning, and representing knowledge at multiple temporal scales. *Artificial Intelligence*, *112*, 181–211.

Theocharous, G. & Mahadevan, S. (2002). Approximate planning with hierarchical partially observable Markov decision process models for robot navigation. In *IEEE Conference on Robotics and Automation*.