Planning in Models that Combine Memory with Predictive Representations of State

Michael R. James and **Satinder Singh**
Computer Science and Engineering
University of Michigan
{mrjames, baveja}@umich.edu

Abstract

Models of dynamical systems based on predictive state representations (PSRs) use predictions of future observations as their representation of state. A main departure from traditional models such as partially observable Markov decision processes (POMDPs) is that the PSR-model state is composed entirely of observable quantities. PSRs have recently been extended to a class of models called memory-PSRs (mPSRs) that use both memory of past observations and predictions of future observations in their state representation. Thus, mPSRs preserve the PSR-property of the state being composed of observable quantities while potentially revealing structure in the dynamical system that is not exploited in PSRs. In this paper, we demonstrate that the structure captured by mPSRs can be exploited quite naturally for stochastic planning based on value-iteration algorithms. In particular, we adapt the incremental-pruning (IP) algorithm defined for planning in POMDPs to mPSRs. Our empirical results show that our modified IP on mPSRs outperforms, in most cases, IP on both PSRs and POMDPs.

Introduction

The problem of finding optimal plans for stochastic dynamical systems modeled as partially observable Markov decision processes (POMDPs) has proved to be a very difficult problem, even for relatively simple domains (Coutilier & Poole, 1996). Algorithms for solving general (unstructured) POMDPs have traditionally been based on value iteration; currently the state of the art general-purpose algorithms are variants of the incremental pruning (IP) algorithm (Cassandra, Littman, & Zhang, 1997). Alternatively, methods for solving POMDPs by searching directly in the policy space have gained popularity (Ng & Jordan, 2000), and there are also methods that make use of special structure in dynamical systems (Coutilier & Poole, 1996). In this paper, we present a planning method that falls into the first category, but is distinguished from existing methods in that it uses a novel representation of dynamical systems in order to improve the performance of the value-iteration approach. This representation, called memory-PSRs or mPSRs (James, Singh, & Wolfe, 2005), builds on recent work on predictive state representations (PSRs) that use predictions of future observations as their representation of state (Littman, Sutton, & Singh, 2001). Memory-PSRs combine memory of past observations with predictions of future observations in their state representation. The main departure from POMDP representations that use hidden or latent state variables is that in *both* PSRs and mPSRs the state is composed entirely of observable quantities.

In this paper, we show that the use of memory in mPSRs reveals additional structure in dynamical systems not captured in PSRs and that this structure can be exploited for planning. In particular, we adapt the IP algorithm already defined for both POMDPs and PSRs to the new mPSRs. Our empirical results show that our modified IP on mPSRs outperforms, in most cases, IP on both PSRs and on POMDPs.

Background material

In this section, we briefly review PSRs, mPSRs, and IP. More complete descriptions are found in Singh, James, & Rudary (2004), James, Singh, & Wolfe (2005), and Cassandra, Littman, & Zhang (1997) respectively.

PSRs and mPSRs

PSRs and mPSRs are both classes of models that use predictive representations of state. These models are distinguished from traditional hidden-state-based models because their state representation is a vector of predictions of the outcomes of *tests* that may be performed on the dynamical system. A test $t = a_1o_1, ...a_ko_k$ is a sequence of alternating actions ($a_i \in \mathcal{A}$) and observations ($o_j \in \mathcal{O}$). Of course, the *prediction* of a test is dependent on the *history*: the actions and observations that have occurred so far; the prediction of a test t at history h is $p(t|h) = prob(o_1, ...o_k|ha_1, ...a_k)$, i.e., the conditional probability that the observation sequence occurs, given that the action sequence is taken after history h. Note that rewards (assumed to take on a finite number of values) are treated as just a special dimension of the general observation space $\mathcal{O}$ in the definition of PSRs above.

PSRs The set of tests, Q, whose predictions constitute a PSR's state representation are called the *core tests*. These core tests are special because at any history, the predictions for *any* test can be computed as a *constant* linear function of the predictions of the core tests. The predictions of the core tests are stored in a $(n \times 1)$ vector called the *prediction vector* $p(Q|h)$, where $n = |Q|$ is called the *dimension* of the dynamical system. In PSRs the prediction vector is the counterpart to belief-states in POMDPs and the last k observations in k-order Markov models. In addition to the set of core tests, a PSR model has *model parameters*: a set of $(n \times n)$ matrices M_{ao}, and $(n \times 1)$ vectors m_{ao}, for all a, o. The model parameters are used to update the state as actions are taken and observations occur, to calculate the predictions of tests, and as shown below are also used in the calculation of policy trees for stochastic planning. For instance, the probability of observation o given that action a was taken at history h is $prob(o|ha) = p(Q|h)^T m_{ao}$.

The immediate expected reward for action a at any history h, denoted $R(h, a)$, is computed as

$$\begin{aligned} R(h,a) &= \sum_r r \cdot prob(r|h,a) \\ &= \sum_r r \sum_{o \in \mathcal{O}^r} p(Q|h)^T m_{ao} \\ &= p(Q|h)^T \sum_r \sum_{o \in \mathcal{O}^r} r \cdot m_{ao}. \qquad (1) \end{aligned}$$

where $\mathcal{O}^r$ is the set of observations in which the reward component takes on value r. Thus, we can define a $(n \times 1)$ vector r^a as

$$r^a = \sum_r \sum_{o \in \mathcal{O}^r} r \cdot m_{ao} \qquad (2)$$

such that $R(h, a) = p(Q|h)^T r^a$. The set of vectors r^a for all $a \in \mathcal{A}$ is used in specifying the PSR model for planning purposes.

mPSRs The mPSR representation (James, Singh, & Wolfe, 2005) is closely related to PSRs in that its state representation contains a vector of predictions of tests, but it also contains a *memory* of the recent past (e.g. the most recent observation, or recent action/observation sequence). In this paper we will only consider memories that are the most recent observation. Let $\mu_1 \dots \mu_m$ represent the m distinct length-1 memories. Each memory will have an associated set of μ-core tests $Q^{\mu_1} \dots Q^{\mu_m}$ respectively. Let the memory at history h be denoted $\mu(h)$. Then the mPSR state at history h is denoted by $[\mu(h), p(Q^{\mu(h)}|h)]$. Note that the number of μ-core tests can be different for each memory, and can never be larger but can be much smaller than the number of core tests for the PSR representation of the system,

For each memory, μ_i, we will keep a set of update matrices $M_{ao}^{\mu_i}$ and vectors $m_{ao}^{\mu_i}$ for all a, o. The update parameters $M_{ao}^{\mu(h)}$ must transform the current prediction vector that makes predictions for μ-core tests $Q^{\mu(h)}$ in history h to the prediction vector for μ-core tests $Q^{\mu(hao)}$ in history hao. Under our assumption of memories being the last observation, all histories belonging to memory μ_i will transition to the same memory μ_j for action-observation pair ao, i.e., j is uniquely determined by i and the pair ao. Thus one can define the state update for mPSRs as follows: upon taking

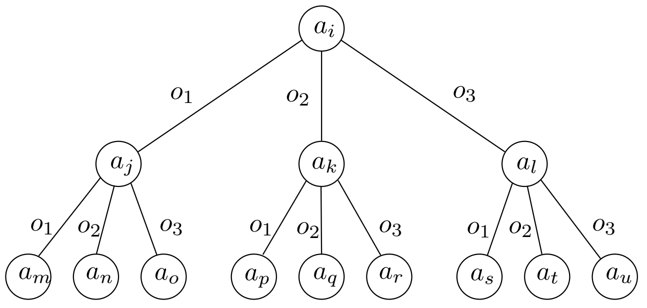

Figure 1: A 3-step policy tree for a dynamical system with three observations.

action a in history h and observing o

$$p(Q^{\mu_j}|hao) = \frac{p(aoQ^{\mu_j}|h)}{p(ao|h)} = \frac{p(Q^{\mu_i}|h)^T M_{ao}^{\mu_i}}{p(Q^{\mu_i}|h)^T m_{ao}^{\mu_i}} \qquad (3)$$

where $\mu(h) = \mu_i$ and $\mu(hao) = \mu_j$. The matrix $M_{ao}^{\mu_i}$ is of size $(|Q^{\mu_i}| \times |Q^{\mu_j}|)$ and the vector $m_{ao}^{\mu_i}$ is of size $(|Q^{\mu_i}| \times 1)$. We note here that a vector equivalent to the PSR immediate reward vector (Equation 2) is defined for mPSRs, and is dependent on both the memory μ and action a as follows

$$r^{\mu,a} = \sum_r r \cdot m_{ar}^{\mu}. \qquad (4)$$

Landmark memories A special case arises when there is a memory for which only one μ-core test is required. Such a memory serves as a *landmark* and the prediction of its μ-core test is constant at all histories corresponding to the landmark. This property of landmarks was exploited in James, Singh, & Wolfe (2005) for tracking dynamical systems with mPSRs. Here we show how our planning method for mPSRs can take advantage of landmarks.

Planning with PSRs

All the planning algorithms we discuss are based on value iteration which proceeds in stages. The value function at stage i is denoted V_i. A dynamic programming update transforms V_i to V_{i+1}, taking into account the one-step system dynamics and one-step rewards. In the limit, as $i \to \infty$, the value function V_i will approach the optimal value function, but typically value iteration is concluded when the difference between two successive value function approximations is sufficiently small.

For most value iteration algorithms (for both POMDPs and PSRs), the value function V_i is represented as a set S_i of parameter vectors for policy trees (Littman, 1996). The parameter vector corresponding to policy tree ρ is denoted w_ρ. Thus, the dynamic programming update transforms the set S_i to the set S_{i+1}. An example policy tree ρ is shown in Figure 1. A policy tree defines a policy consisting of an initial action and, for each observation, another (one step shorter) policy tree. In James, Singh, & Littman (2004) it is shown that the expected discounted reward for following ρ's policy when at prediction vector $p(Q|h)$ is given by $p(Q|h)^T w_\rho$. In other words, the value of ρ is a linear function of the prediction vector, and the value function V_i is a piecewise linear function defined by the upper surfaces of these functions for all $w_\rho \in S_i$. *The upper surface only makes use of some of the vectors $w_\rho \in S_i$, and a key step in improving the performance of these algorithms is to purge the set S_i so that it contains as few vectors as possible, given that it must represent the value function.*

This is analogous to the situation for POMDPs: the value for ρ is a linear function of the belief state, and the value function is a piecewise linear function of the belief state defined by the upper surfaces for all $w_\rho \in S_i$. Therefore, the stochastic planning algorithms for POMDPs can be applied to PSRs with only slight modifications. The incremental pruning (IP) algorithm (Zhang & Liu, 1996) has emerged as one of the fastest, and we present a version of it for PSRs (called PSR-IP) next.

Incremental pruning The PSR-IP algorithm transforms the set S_i to S_{i+1} via a series of intermediate sets. The following vector notation is used: vector sums are componentwise, and we define the cross sum of two sets of vectors: $A \oplus B = \{\alpha + \beta | \alpha \in A, \beta \in B\}$. Given a set S_i, there are two intermediate sets used to calculate S_{i+1}. They are

$$S_o^a = purge(\{\tau(w_\rho, a, o) | \forall w_\rho \in S_i\}) \quad (5)$$

$$S^a = purge\left(\bigoplus_o S_o^a\right) \quad (6)$$

and the new set is

$$S_{i+1} = purge\left(\bigcup_a S^a\right) \quad (7)$$

where $\tau(w_\rho, a, o)$ is the $(n \times 1)$ vector given by

$$\tau(w_\rho, a, o) = r^a/|\mathcal{O}| + \gamma M_{ao} w_\rho \quad (8)$$

where the *purge* routine (also called pruning or filtering) uses linear programming to take a set of vectors and return only those vectors necessary to represent the upper surface of the associated value function. The efficiency of this routine is very sensitive to the size of the associated set, so reducing the size of the associated sets is critical. The efficiency of IP is obtained by reducing the size of these sets wherever possible, even at the expense of pruning more often.

For IP, transforming the set S_i to the set S_{i+1} is accomplished in Equations 5, 6, and 7. The sets in Equations 5 and 7 are constructed in a straightforward manner, while Equation 6 makes use of the fact that

$$purge(A \oplus B \oplus C) = purge(purge(A \oplus B) \oplus C). \quad (9)$$

This incremental purging of S^a is the key to the performance benefits of incremental pruning

To adapt POMDP-IP to PSRs, there is one modification that must be made, which involves identifying *valid* prediction vectors. For example, in POMDPs any correctly sized vector of positive numbers that sums to 1.0 is a valid POMDP belief state. For PSRs, there is no corresponding simple constraint on prediction vectors. However, a number of additional (more complex) constraints are presented in James, Singh, & Littman (2004) that will identify the majority of invalid prediction vectors. For prediction vector p, some constraints are:

1. For every entry p_i of p: $0 \leq p_i \leq 1$.
2. For every action, observation pair a, o, $0 \leq p^T m_{ao} \leq 1.0$.
3. Constrain all one-step extensions of the core tests q^i. This ensures that every next prediction vector would have valid entries. For every core test q^i and action, observation pair a, o : $0 \leq p^T m_{aoq^i} \leq p^T m_{ao} \leq 1$.

Of course, for mPSRs, the constraints must reference memories, so the constraints for identifying prediction vectors for memory μ would use mPSR parameters m_t^μ. The three constraints listed above will be included in the linear program used in the purge routine. The addition of these constraints results in more parameter vectors w_ρ being removed from the sets S_i, improving the performance of the algorithm.

Planning in mPSRs

The memory part of the state representation in mPSRs can be exploited for planning in two ways: 1) to decompose the problem; and 2) to construct policy trees efficiently.

Using memories to decompose the problem

The idea is that each memory will maintain its own policy, i.e., its own set of policy trees. Consequently, instead of pruning one large set of policy trees for all memories at once, the mPSR approach will prune many smaller sets of policy trees, one for each memory. Intuitively, the sets of policy trees for each memory will be smaller than the set of policy trees for the system as a whole because the number of situations that a dynamical system can be in is typically larger than the number of situations that the dynamical system can be in when at a particular memory. Therefore, the number of policy trees needed for a particular memory is fewer than the number of policy trees needed for the system in general.

This is illustrated in Figure 2, where the t-step policy trees are shown as sets (clouds) being used to construct the $t+1$-step policy trees. For PSRs, a single set is maintained and covers all memories, while for mPSRs a different set is kept for each memory (last observation).

We now show that this decomposition maintains an exact representation of the value function; it is not gaining performance by making approximations.

Lemma 1. *For mPSRs, the value of a policy tree is a linear function of the mPSR state, meaning that, when at memory μ, the value of any policy tree is a linear function of the associated prediction vector $p(Q^\mu|h)$.*

Proof. This proof is by induction, which first assumes that the value of t-step policy trees are a linear function of the mPSR state and shows that all $(t+1)$-step policy trees are a linear function of the mPSR state. Then, it is shown that the value of all 1-step policy trees are linear functions of the mPSR state.

(inductive step) Consider a $(t+1)$-step policy tree ρ with initial action a and t-step policy trees ρ^o for every observation o. Assume that the expected discounted reward function V_{ρ^o} for all ρ^o are linear functions of the new prediction vectors (for memory o) and are written $V_{\rho^o}(hao) =$

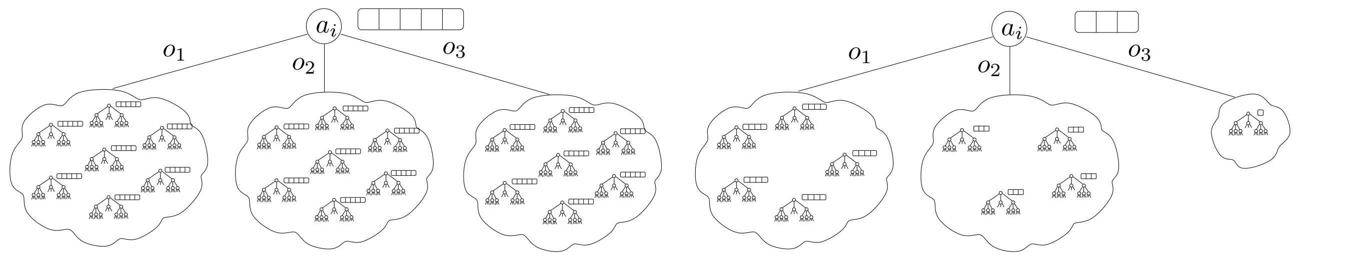

Figure 2: Illustration of the construction of $(t+1)$-step policy trees from t-step policy trees for PSRs (left) and mPSRs (right). For PSRs, the policy tree is for action a_i, while for mPSRs, the policy tree is for memory o_2 and action a_i. Every combination of t-step policy trees results in a new $(t+1)$-step policy tree, so for PSRs, there are 7^3, and for mPSRs there are $5 \cdot 4 \cdot 1$. Also shown are the sizes of the vectors used to calculate the expected reward vector, which is equal to the dimension of the linear program to be solved.

$p(Q^o|hao)^T w_{\rho^o}$ for vectors w_{ρ^o}. The expected discounted reward for tree ρ at history h and memory μ is

$$\begin{aligned}
V_\rho(h) \quad &= \quad R(h,a) + \gamma \sum_{o\in\mathcal{O}} prob(o|a,h) V_{\rho^o}(hao) \\
&= p(Q^\mu|h)^T r^{\mu,a} + \gamma \sum_{o\in\mathcal{O}} \left(p(Q^\mu|h)^T m^\mu_{a,o}\right)\left(p(Q^o|hao)^T w_{\rho^o}\right) \\
&= p(Q^\mu|h)^T r^{\mu,a} + \gamma \sum_{o\in\mathcal{O}} p(Q^\mu|h)^T m^\mu_{a,o} \left(\frac{p(Q^\mu|h)^T M^\mu_{ao}}{p(Q^\mu|h)^T m^\mu_{ao}}\right) w_{\rho^o} \\
&= p(Q^\mu|h)^T r^{\mu,a} + \gamma \sum_{o\in\mathcal{O}} p(Q^\mu|h)^T M^\mu_{ao} w_{\rho^o} \\
&= p(Q^\mu|h)^T \left(r^{\mu,a} + \gamma \sum_{o\in\mathcal{O}} M^\mu_{ao} w_{\rho^o}\right). \qquad (10)
\end{aligned}$$

Therefore,

$$w_\rho = r^{\mu,a} + \gamma \sum_{o\in\mathcal{O}} M^\mu_{ao} w_{\rho^o} \qquad (11)$$

is a $(|Q^\mu| \times 1)$ vector, and $V_\rho(h)$ is a linear function of $p(Q^\mu|h)$ for memory μ.

(initial step) A one-step policy tree ρ specifies a single action, so the expected reward for ρ at memory μ is given by Equation 4, and is a linear function of the prediction vector. □

The practical implication of this lemma is that for each memory a separate set of policy trees may be found that are optimal for that particular memory, and are dependent only on the prediction vector for that memory. Therefore, pruning the policy trees takes place on a smaller dimensional space. However, the drawback is that a separate set of policy trees will need to be obtained for each memory, increasing the number of times that pruning need be done.

Landmarks Landmarks are special because at a landmark the prediction of the (sole) μ-core test takes on a constant value. Therefore, there can only be a single policy tree, or equivalently a single value, to be stored for the landmark (this is reminiscent of and related to planning in MDPs where each nominal-state has a single associated value). Because the prediction is constant, no linear programs need be solved for landmarks; a straightforward maximization over the candidate policy trees may be used instead. This is much faster and we exploit this in our algorithm below whenever landmarks are available.

Using memories to construct the policy tree

We now show how length-one memories can be used advantageously in the construction of policy trees. For instance, take the $(t+1)$-step policy tree in Figure 2. Associated with each observation is a t-step policy tree, which is one of the optimal policy trees from the previous iteration of the planning algorithm. Before pruning, the set of $(t+1)$-step policy trees contains all combinations of t-step policy trees at each observation.

For instance, in PSR (or POMDP) planning, for each observation, any of the optimal t-step policy trees could be used at that observation, so there are (# t-step policy trees)$^{|\mathcal{O}|}$ new $(t+1)$-step policy trees[1] to be pruned.

On the other hand, when planning with mPSRs, each observation corresponds to a memory, and so only the policy trees corresponding to that particular memory must be evaluated. Because each of these sets is typically smaller than the entire set of policy trees, the number of resulting trees is often reduced dramatically, which results in significant computational savings. With one memory for each observation, the total number of resulting policy trees is (# policy trees for memory 1) * (# policy trees for memory 2) * ... * (# policy trees for memory $|\mathcal{O}|$).

Figure 2 illustrates how length-one memories fit naturally into the existing structure of policy trees. Of course, as mentioned previously, the drawback is that a set of $(t+1)$-step policy trees must be constructed for each memory individually. However, this drawback is typically overcome by the benefits.

For example, take the Cheese Maze system used in empirical testing (see Table 1). There are 7 observations, and the PSR has 11 core tests. Using observations as memories, the mPSR has 1,1,1,1,2,2,3 μ-core tests for its 7 memories. During a randomly chosen planning iteration (number 344), the PSR has 16 policy trees to begin with, so the algorithm must prune $16^7 = 268,435,456$ new policy trees (ignoring the effects of IP) on a 11-dimensional space. For mPSRs, the memories at this iteration have 1,1,1,1,1,2,2 policy trees,

[1]IP does some work to reduce this number as policy trees are being constructed, but the basic idea still holds.

so the algorithm must prune 4 policy trees, but must do so seven different times, four times on a 1-dimensional space (no linear programming needed), twice on a 2-dimensional space, and once on a 3-dimensional space. This is a significant improvement.

On the other hand, the Tiger system (see Table 1) has the same number of μ-core tests at each of the two memories as it has for the PSR itself. Using memories reveals none of the structure of the system. So, for each of the two memories, the same amount of pruning must be done as is done for the PSR as a whole, so twice as much computation is required. However, this effect can be mitigated by detecting when memories reveal structure in the dynamical system, and using those results to choose the proper memories (or lack thereof).

Implementing mPSR-IP

Adapting the PSR-IP algorithm to mPSRs involves three changes to the algorithm. First, the sets $S_o^{\mu,a}$, $S^{\mu,a}$, and S_i^{μ} must be maintained separately for each memory μ. Essentially, incremental pruning is done for each memory individually.

Secondly, the calculation of $S_o^{\mu,a}$ from $\tau(w_\rho, a, o)$ (Equation 5) need only consider the $w_\rho \in S_i^o$ for the memory corresponding to observation o. The calculation of $\tau(w_\rho, a, o)$ is just a modification of Equation 8 to include memories, so

$$\tau(w_\rho, a, o) = r^{\mu,a}/|\mathcal{O}| + \gamma M_{ao}^{\mu} w_\rho. \qquad (12)$$

The calculations of $S^{\mu,a}$ and S_i^{μ} are defined just as in Equations 6 and 7, but with the addition of memories.

The third change is that landmarks are treated specially: there need be only a single policy tree for each landmark, and so the optimal policy tree can be found without linear programming. If the constant prediction vector $p(Q^L)$ for each landmark L is precomputed, then the optimal policy tree is just: $\mathrm{argmax}_\rho(p(Q^L) \cdot w_\rho)$.

Empirical work

In order to evaluate the effectiveness of incremental pruning on mPSRs (mPSR-IP), we compared it against both PSR-IP and POMDP-IP, using a suite of standard POMDP dynamical systems available at Cassandra (1999). To this suite, we added one problem of our own making, the 4x3CO system in Table 1. This problem was constructed so that all its observations are landmarks and was designed to show off mPSR-IP's ability to exploit landmarks in an extreme case. For each of the test systems, Table 1 also lists important characteristics such as the number of nominal-states in the POMDP model, the number of core tests in the PSR model, and the number of μ-core tests for each memory in the mPSR model.[2]

[2]Note that mPSR-IP was modified slightly to detect when the use of memory did not reveal any structure in the system, by checking whether the number of μ-core tests for every memory μ was equal to the number of PSR core tests. In these cases (Tiger and Paint), the performance of mPSR-IP could only be worsened by using memories, so the algorithm executed PSR-IP instead.

Below we compare PSR-IP and mPSR-IP to understand the relative benefit to planning of adding memory to the state representation. Furthermore, because POMDP-IP generally outperforms PSR-IP, mPSR-IP and POMDP-IP are compared below in order to evaluate whether mPSR-IP is a useful improvement on methods for stochastic planning.

The experimental procedure was as follows. The IP algorithms were run for a maximum of 500 iterations, or when an 8 hour (28800 seconds) time limit was reached. Care was taken to evaluate all runs under comparable conditions. The results in Table 1 show the number of iterations completed within the eight hour time limit, and if all 500 iterations were completed it also shows the total amount of time (in seconds) to complete those 500 iterations. For all three algorithms, it is better to have a greater number of iterations completed in terms of the quality of the value function found, and if the limit of 500 iterations was reached, it is better to have a shorter execution time.

Evaluating the use of memory

The first evaluation is how the use of memory in mPSRs affects the performance of planning. To do this, we compare the results for PSR-IP and mPSR-IP.

For the systems on which both algorithms completed 500 iterations, the timing results show that mPSR-IP outperforms PSR-IP by a significant margin on all systems except for Tiger and Paint, for which memories revealed no structure. For the last two systems listed in Table 1, mPSR-IP outperforms PSR-IP significantly. For Shuttle, PSR-IP only completes 8 iterations in 8 hours, while mPSR-IP finishes 500 in just over 4 hours. For the 4x3 system, PSR-IP completes 9 iterations while mPSR-IP completes 20. While this is only about twice as many iterations, the significance of this improvement is evident when considering that each iteration is significantly more difficult than the previous. For instance, for PSR-IP the first nine iterations together took about 63 minutes, but the 10^{th} iteration (not included in Table 1 because it went beyond the time limit) **alone** took 8 hours and 52 minutes. Because of the increases in computational requirements between consecutive iterations, obtaining twice as many iterations is a significant improvement. Finally, observe that for the 4x3CO system, mPSR-IP very significantly outperforms PSR-IP as expected.

Therefore, for systems where memories reveal some of the system structure, using mPSRs for planning has significant performance benefits over using PSRs for planning. This is evident even when the revealed structure (measured by number of μ-core tests) is only significant for one memory (4x4), and even when the revealed structure is not great for any memory (Network).

Evaluating POMDP and mPSR planning

Although comparing mPSR-IP to PSR-IP demonstrates how adding memories to predictions of the future in the state representation can speed up the algorithm, our larger goal is to develop an algorithm that outperforms POMDP-IP. Note that POMDP-IP generally outperforms PSR-IP (see Table 1).

In evaluating these algorithms, the Tiger and Paint systems have the same problem as above: memories do not re-

Table 1: Test Systems and Results of IP on POMDPs, PSRs, and mPSRs

System	POMDP	PSR	mPSR	Stages			Time (sec.)		
	nominal-states	tests	tests	POMDP	PSR	mPSR	POMDP	PSR	mPSR
1D maze	4	4	1, 3	500	500	500	2	6	3
Tiger	2	2	2, 2	500	500	500	49	88	88
Paint	4	4	4, 4	500	500	500	51	190	190
Cheese	11	11	1,1,1,1,2,2,3	500	500	500	21	528	27
4x3CO	11	11	1,1,1,1,1,1,1,1,1,1,1	500	500	500	18	2051	<1
4x4	16	16	1, 15	500	500	500	16	3782	486
Network	7	7	4, 6	500	500	500	8237	117	15
Shuttle	8	7	1,1,2,2,4	213	8	500	n/a	n/a	15409
4x3	11	11	1,1,1,1,3,4	10	9	20	n/a	n/a	n/a

veal any structure, and the resulting performance of mPSR-IP is worse than POMDP-IP. However, for the remaining systems, the results for mPSRs are better. For two of the systems, the performance of POMDP-IP and mPSR-IP is roughly equivalent, and for one system, POMDP-IP outperforms mPSR-IP, but for the other four systems, mPSR-IP outperforms POMDP-IP, often by an order of magnitude. In fairness, it must be reported that POMDP-IP on the Shuttle system was progressing quickly when the time limit was reached, and would have completed 500 iterations within 9 hours (and so mPSR-IP was roughly twice as fast). The results of the 4x3 system show the most significant gains. The number of iterations for mPSR-IP is double that of POMDP-IP, but again, typically each iteration of the algorithm is significantly more difficult than the previous. For instance, the 11^{th} iteration of POMDP-IP did not complete within 36 hours.

Conclusion

We proposed an extension of the incremental pruning algorithm to mPSRs and showed how the use of memory as part of the state representation is exploited both to decompose the value iteration calculation as well as in the construction of the associated policy trees. We also demonstrated how memories that serve as landmarks can be used for particularly efficient planning. Our empirical results demonstrate that generally in dynamical systems for which the mPSR model is more compact than the PSR or POMDP model, i.e., systems for which the memory part of mPSRs captures structure in the system, our mPSR-IP algorithm outperforms both the PSR-IP and POMDP-IP algorithms.

As future work we will explore the automatic identification of variable-length memories that lead to good planning performance instead of using fixed-length memories as in the current work. This may lead to greater and more consistent benefit over existing methods than demonstrated here.

Acknowledgment The research reported in this paper was supported by NSF grant IIS-0413004.

References

Cassandra, A.; Littman, M. L.; and Zhang, N. L. 1997. Incremental Pruning: A simple, fast, exact method for partially observable Markov decision processes. In *13th Annual Conference on Uncertainty in Artificial Intelligence (UAI–97)*, 54–61.

Cassandra, A. 1999. Tony's pomdp page. http://www.cs.brown.edu/research/ai/ pomdp/index.html.

Coutilier, C., and Poole, D. 1996. Computing optimal policies for partially observable markov decision processes using compact representations. In *13th National Conference on Artificial Intelligence*.

Izadi, M. T., and Precup, D. 2003. A planning algorithm for predictive state representations. In *Eighteenth International Joint Conference on Artificial Intelligence*.

James, M. R.; Singh, S.; and Littman, M. L. 2004. Planning with predictive state representations. In *The 2004 International Conference on Machine Learning and Applications*.

James, M. R.; Singh, S.; and Wolfe, B. 2005. Combining memory and landmarks with predictive state representations. In *The 2005 International Joint Conference on Artificial Intelligence*.

Littman, M. L.; Sutton, R. S.; and Singh, S. 2001. Predictive representations of state. In *Advances In Neural Information Processing Systems 14*.

Littman, M. L. 1996. Algorithms for sequential decision making. Technical Report CS-96-09, Brown University.

Ng, A. Y., and Jordan, M. 2000. PEGASUS:A policy search method for large MDPs and POMDPs. In *16th Conference on Uncertainty in Artificial Intelligence*, 406–415.

Singh, S.; James, M. R.; and Rudary, M. R. 2004. Predictive state representations, a new theory for modeling dynamical systems. In *20th Conference on Uncertainty in Artificial Intelligence*.

Zhang, N. L., and Liu, W. 1996. Planning in stochastic domains: Problem characteristics and approximation. Technical Report HKUST-CS96-31, Hong Kong University of Science and Technology.