Approximate Value Iteration on Predictive State Representations

Thesis for graduation as Master of Artificial Intelligence

Ton Wessling

Informatics institute, Faculty of Science

University of Amsterdam

twesslin@science.uva.nl

Supervised by

Nikos Vlassis

December 17, 2005

Abstract

Predictive State Representations (PSRs) are models of discrete-time dynamical systems and propose an alternative to traditional models such as Partially Observable Markov Decision Processes (POMDPs). PSRs and POMDPs are very much alike when it comes to planning with uncertainty. They are applicable on the same type of problems, but the difference is that POMDPs use a hidden state model, whereas PSRs only use 'tests', i.e. sequences of actions and observations to model the world, which are truly observable and much more tangible. In this work we will show that PERSEUS, a randomized point-based value iteration algorithm (by Spaan & Vlassis, [1]) for computing approximate optimal policies in POMDPs, can also be used for finding approximate optimal policies for PSRs with only little modifications. The resulting algorithm, PERSEUS$_{PSR}$ achieves the same results as the exact PSR-IP algorithm of James, Singh & Littman [3], for the same problems in much less time. Other well-known POMDP problems such as Hallway and Tag have also been transformed to equivalent PSR problems, and in this thesis it will be shown that PERSEUS$_{PSR}$ is a very fast approximate value iteration algorithm which produces an outcome policy that achieves the same results in experiments as one from an exact algorithm, such as PSR-IP, would achieve.

Contents

1 Introduction 3

2 Partially Observable Markov Decision Processes 5

3 Value iteration on POMDPs 7

4 PERSEUS: Randomized point-based value iteration on POMDPs 10

5 Predictive State Representations 13

6 Value iteration on PSRs 16

7 PERSEUS$_{PSR}$: Randomized point-based value iteration on PSRs 19

8 Transforming a POMDP to a PSR 21

9 Results 23

10 Conclusion 34

11 Bibliography 35

12 Acknowledgements 36

1 Introduction

Imagine yourself in a large hallway with five doors, four of them yellow and one of them green. You cannot see far from your position and you want to go to the green door. Having a map, you could decide what to do if you knew where you are. The problem is, you don't know where you are in that hallway, but at least you can observe you're not next to one of the doors. To determine where you are, you decide to go a little down the hall. After a few steps, you arrive at a yellow door. Now you still don't exactly know where you are, but at least you know you are next to a yellow door - you are narrowing the amount of possible locations you might be at. At some point, you have walked a little more down the hall and you have found the green door or the fourth yellow door. At that point you exactly know where you are, since there's only one green door or you have seen all the yellow doors. Now you can decide with 100% certainty what to do to find the green door. This entire idea can be modeled as a Partially Observable Markov Decision Process (POMDP): An agent is in a certain, unknown state (an unknown position in the corridor), and by performing actions it gets observations from its surroundings. Using these observations and actions, the agent can determine the exact state it is in with more and more certainty, and therefore plan more efficiently to achieve its goal.

A Predictive State Representation (PSR) can be used for the same kind of problem, but an agent with a PSR does not use states for world representation, it makes use of probabilities on receiving observations in the future, as a result of taken actions and the sofar perceived history. In the hallway case, a PSR would use the probabilities of seeing a yellow door, a green door, the end of the hall and perhaps more observations such as seeing a computer. Of course, the more one goes down the hall, the chance of observing yellow doors or the green door changes over time, and the probability of observing a computer will most likely remain close to zero (as there are usually no computers in the corridor). As soon as you have seen a yellow door, the chance of seeing another yellow door will be low for a while, until you have taken a few steps (there will most likely be some space between the doors) and then the probability of seeing a yellow door rises again. Using this setting, an agent tries to maximize the probability that a certain (set of) action(s) lead to the observation of the goal state, the green door.

The main difference between these two approaches is that PSRs only use 'tangible' and really observable quantities, while POMDPs would also use an abstract and hidden state model, a sort of 'map' of the world. If one knows what one can do in an unknown world, and what the observations are, a PSR model could work there, as for a POMDP the entire state space (the world) needs to be known beforehand as well. In both settings, the important part is that the agent must perform a certain task as good as possible, and to do that the agent needs to plan ahead on what to do. An optimal (computed) policy defines the optimal action to perform in each situation, either for a PSR or a POMDP, such that the agent fulfills its task as good and as quickly as possible.

This thesis will first explain what POMDPs are, in mathematical terms and how POMDPs are generally implemented. After that, value iteration and approximate value iteration on POMDPs will be treated as well as an algorithm called PERSEUS to solve a POMDP using approximate value iteration. This entire framework of POMDPs is treated to enable the reader to see the parallels and differences between POMDPs and PSRs, which will be explained subsequently. These similarities and differences reside in the definition of a PSR as well as in the (approximate) value iteration. Using this knowledge, the PERSEUS algorithm for solving POMDPs using approximate value iteration can be transformed to apply to PSRs instead of POMDPs. The results of this conversion will be shown and compared to other results in this field of work, yielding a conclusion.

2 Partially Observable Markov Decision Processes

A POMDP is defined by a tuple $< S, A, O, R, \mathcal{T}, \mathcal{O}, \mathcal{R}, b_0 >$ in which S is the set of al world states. In a POMDP, the agent keeps track of a probability vector over all possible world states s, called a belief vector b, which is initialized as b_0 being

$$b_0(s) = \frac{1}{|S|}, \forall s \in S$$

because the agent does not know what the true state is; all are equally likely at the start. The probability vector b is updated every timestep, because every timestep the agent takes an action $a \in A$ and receives an observation $o \in O$. This means some states become less likely to be true and other states become more likely to be true as a result of this action and observation, as time keeps on ticking. The updating of the belief vector to achieve this probability updating is done by Bayes' Rule:

$$b_a^o(s') = \frac{p(o|s',a)}{p(o|a,b)} \sum_{s \in S} p(s'|s,a) b(s) \tag{1}$$

in which $p(o|a,b) = \sum_{s' \in S} p(o|s',a) \sum_{s \in S} p(s'|s,a)b(s)$ is a normalizing constant to make sure $\sum_s b(s) = 1$. This formula calculates the new probability distribution over all states, using the probability that the agent ends up in state s' after performing action a in state s, along with the probabilities of the possible observations which can be made. When implemented, often matrices $\mathcal{T}$ and $\mathcal{O}$ are used to perform this calculation. Transformation matrix $\mathcal{T}^a$ is a collection of $|S| \times |S|$ matrices for each $a \in A$, where each entry $T_{i,j}^a$ denotes the probability of the transition of s_i to s_j by performing action a. The $\mathcal{O}$ set of matrices, called the observation matrices, is a set of $|S| \times |S|$ diagonal matrices $\mathcal{O}^{a,o}$ for every $a \in A$ and $o \in O$. The entry $\mathcal{O}_{i,i}^{a,o}$ is the probability of observing observation o as a result of performing action a in state i. Bayes' Rule (1) then reads:

$$b_a^o(s') = \frac{\mathcal{O}_{s',s'}^{a,o} \sum_{s \in S} \mathcal{T}_{s,s'}^a b(s)}{\sum_{s' \in S} \mathcal{O}_{s',s'}^{a,o} \sum_{s \in S} \mathcal{T}_{s,s'}^a b(s)}$$

To use the POMDP framework for planning, reward needs to be incorporated too — an agent needs to know whether an action is a good one to perform in a certain state or not. This means the rewards are results of actions performed in states according to $r(s,a)$. So the same action in a different state will most likely give different rewards. If the agent must fulfill its task as good and as quickly as possible, it should of course perform the optimal action in every state. To find an optimal action to perform in each state, one needs to search for the action which gives highest reward in each state. To compute an optimal policy V^*, which defines the optimal action to perform, the rewards for the current action need to be known, and one needs to look ahead a few steps instead of just one. For looking ahead one multiplies the reward of such a future state with the probability that the agent will be in that state, at the timestep concerned. When one works with beliefs, the above arguments transfer to belief states, and V^* is the optimal value function over all beliefs. This V^* satisfies the Bellman optimality equation $V^* = HV^*$(H is the Bellman backup operator), or

$$V^*(b) = \max_a \left[\sum_{s \in S} r(s,a) b(s) + \gamma \sum_{o \in O} P(o|a,b) V^*(b_o^a) \right] \tag{2}$$

where $V^*(b)$ is the value of the value function in belief point b. The optimal action to perform in b is the action which maximizes $V^*(b)$. The first part of 2 is clearly the immediate reward, and the second part the expected value for the remaining timesteps, which, iteratively, depend on a timestep one step further in the future. This way it is possible to determine the number of steps to look ahead and the importance of those future steps, represented by discount factor γ: the further in the future, the less important it becomes. Usually, one sets γ to a value close to 1, i.e. 0.95.

3 Value iteration on POMDPs

Value iteration has as goal to find an improved value function as a function of the old one, $V_{n+1}(b) = f(V_n(b))$. There are multiple ways to do value iteration on POMDPs. Most well-known and much-used methods use the characteristic shape of the value function. The value function V^* of POMDPs has a PieceWise Linear Convex (PWLC) shape, consisting of multiple vectors (or hyperplanes for dimensions > 2). A sample picture of a (two-state POMDP case) PWLC value function is shown in figure 1 below.

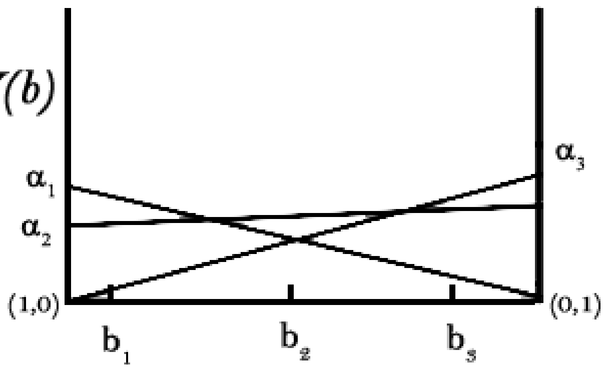


Figure 1: a sample PWLC value function.

All the vectors of the value function are each linked to an action, and the vector α_i which maximizes $V^*(b) = (\alpha_i \in V^*) \cdot b$ therefore denotes the optimal action to perform in belief point b. If one looks several steps ahead, the value function might change; an action which might look like a good one to perform at a certain timestep may prove to be very disadvantageous later on. Therefore, the value function needs to be improved. The updating of the vectors of the value function is done by the backup-operator:

$$backup(b) = \arg\max_a b \cdot \left[r_a + \gamma \cdot \sum_o \arg\max_i b \cdot \sum_{s'} P(o|s', a) P(s'|s, a) \alpha_i(s') \right] \qquad (3)$$

in which $\alpha_i(s')$ is a vector of the current value function. $backup(b)$ computes the optimal vector for a given belief b by back-projecting all vectors in the current horizon value function one step from the future and returning the vector that maximizes the value of b. $backup(b)$

is equal to computing α_{n+1}^b, which is the updated vector which optimizes (or at least improves) the value of belief point b for timestep n+1.

Exact value iteration as in straightforward pruning (calculating all possible vectors and then one by one eliminating unnecessary vectors) has been sped up in [6]. This algorithm also makes efficient use of "witness regions", which are in fact the sets of belief points of which the value function is maximized by the same vector; as an example, figure 1 has only three witness regions of which b_1, b_2 and b_3 are the witness points. This method, however, was not as fast as the incremental pruning algorithm, called POMDP-IP ([5]), which breaks the value function down into simpler combinations, where b_{ao} is the updated belief vector:

$$\begin{array}{lll} V_{n+1}(b) & = \max_{a \in A} V_n^a(b) & = \max_{\alpha \in W_{n+1}} b \cdot \alpha \\ V_n^a(b) & = \sum_{o \in O} V_n^{ao}(b) & = \max_{\alpha \in W^a} b \cdot \alpha \\ V_n^{ao}(b) & = \frac{\sum_{s \in S} r_a(s) P(s)}{|O|} + \gamma P(o|b,a) V(b_{ao}) & = \max_{\forall \alpha \in W^{ao}} b \cdot \alpha \end{array}$$

Earlier, the value function was expressed as $V^*(b) = (\max_\alpha \alpha_i \in V^*) \cdot b$, which means we can express the partial value functions as shown in the rightmost part. W is in fact the S from [5], but this has been changed to W avoid confusion with the set of all possible world states which one finds throughout this thesis. All the W-sets can be described as

$$W_{n+1} = purge\left(\bigcup_{a \in A} W^a\right) \tag{4}$$

$$W^a = purge\left(\bigoplus_{o \in O} W^{ao}\right) \tag{5}$$

$$W^{ao} = purge(\tau\alpha, a, o)|\alpha \in W) \tag{6}$$

in which $\tau(\alpha, a, o)$ is a $|W|$-sized vector given by

$$\tau(\alpha, a, o)(s) = (1/|O|) r_a(s) + \gamma \sum_{s' \in S} \alpha(s') P(o|s', a) P(s'|s, a)$$

where the reader can probably recognize much with respect to section 2. The Purge routine is in fact a filtering step, removing all the vectors which will definitely not contribute to the final V: it reduces the sets to their unique minimum form, being a PWLC form. Formulae 4 and 6 can be computed very efficiently, but 5 grows exponentially in $|O|$, and is computed in the most efficient way using the fact that $purge(A \oplus B \oplus C) = purge(purge(A \oplus B) \oplus C)$. Still, this produces $|A||V_n|^{|O|}$ vectors at each value iteration step and most of these vectors are useless. The subsequently identifying and pruning needs linear programming, and is therefore very costly in high dimensions.

More speed-ups have been investigated, as interleaving approximate iteration steps and exact value iteration steps, but a fully approximate method like PERSEUS, described in the next section, is currently by far the fastest.

4 PERSEUS: Randomized point-based value iteration on POMDPs

Because of the PWLC shape of the value function, one can do approximate planning in a way that improving the value function for one belief point, the value function for a subset of the other belief points is also improved, more or less what happens with the "witness points". This has been done by Vlassis & Spaan [1], in their algorithm called PERSEUS. The main scheme of PERSEUS is as follows:

1. Randomly collect a set B of reachable belief points and initialize the value function V_0 as one vector with all its components minimal, to $\frac{1}{1-\gamma}\min_{s,a} r(s,a)$. Starting with V_0, PERSEUS will perform value function updates in step 3-5 until some convergence criterion is met.
2. Set $V_{n+1} = \emptyset$. Initialize $\tilde{B}$ to B, where $\tilde{B}$ is the set of non-improved belief points.
3. Sample a belief point b uniformly at random from $\tilde{B}$, and compute $\alpha = backup(b)$.
4. If $b \cdot \alpha \geq V_n(b)$ then add α to V_{n+1}, otherwise add $\alpha' = \arg\max_{\alpha^i \in V_n}$ to V_{n+1}.
5. Compute $\tilde{B} = \{b \in \tilde{B} : V_{n+1}(b) - \epsilon < V_n(b)\}$ and go to step 2 if $\tilde{B}$ is not empty yet.

PERSEUS applies the backup operator on a randomly sampled belief point b, and when the newly found vector (as an outcome of the backup-operator) of the value function gives a sufficiently (due to the ϵ value, set to $1E^{-8}$) higher value than one or more of the old vectors, the new vector will probably improve some other values of belief points in the vicinity of b as well. There are two reasons for using the ϵ parameter; If this value is not present or equal to machine precision, it might happen that PERSEUS finds an improved vector for belief point b, but that b is not removed from $\tilde{B}$. Therefore, a very small value for ϵ will take care of this problem. The other reason is that if the value for b is updated very little, the algorithm can be forced a little to find a better vector for b. If one sets ϵ to a very large value, i.e. 0.5, the number of vectors which make up V is greatly influenced and diminished, resulting in a less optimal policy. An optimal value for this ϵ could be searched for, but $1E^{-8}$ works fine - it is small enough to have no impact on the outcome of PERSEUS, and significantly larger than machine precision, which would otherwise lead to problems. The PERSEUS value iteration is best explained by figure 2, which is an illustration of a two-state case of value iteration by PERSEUS. In this

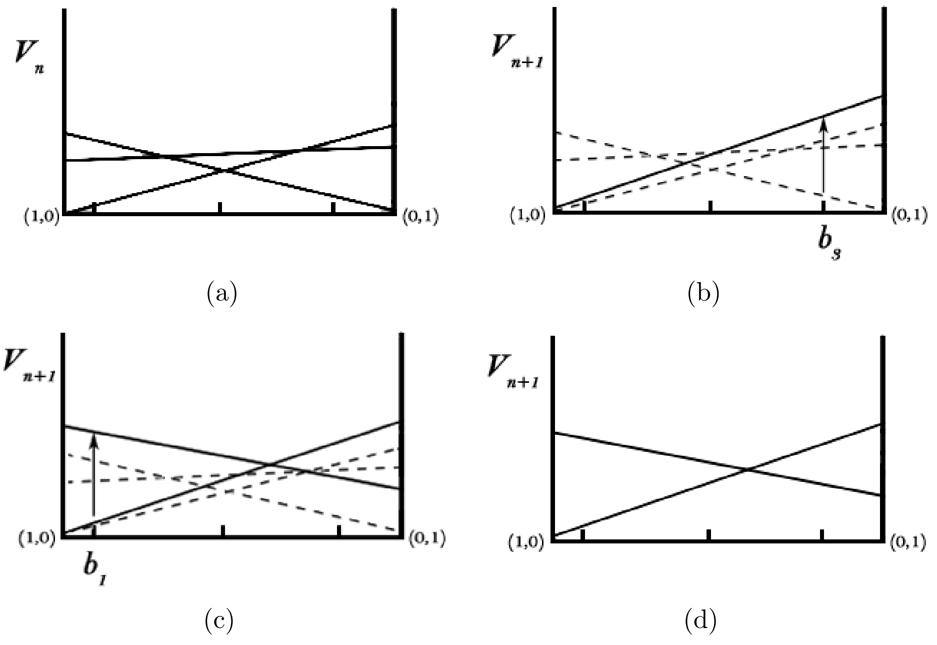


Figure 2: A typical backup-procedure by PERSEUS in a two-state POMDP.

figure, the belief space is on the x-axis and the y-axis represents $V(b)$. Solid lines compose the value function in the current stage, n, and dashed lines represent the updated value function. The computation of V_{n+1} from V_n in figure 2 follows the steps 3 to 5 of the main scheme of PERSEUS. In 2(a) one sees the value function in stage n; 2(b) shows belief point b_3 being randomly sampled and $\alpha = backup(b_3)$ is added to V_{n+1} which also improves the value of b_2; 2(c) displays the sampling of b_1 (which was the last remaining belief point) and the computation of $\alpha = backup(b_1)$, which is also added to V_{n+1}. The last part, 2(d), shows that all belief points have been updated and PERSEUS is done for stage $n + 1$. Of course, one can keep iterating and updating v_n for a number of iterations or until a convergence criterion has been met.

After a number of iterations, all belief points have been updated by improving only a relatively small amount of vectors. Of course, this needs to be done for every timestep in the future span that is to be concerned. When a certain number of future timesteps have been taken into account as well, the improving of the value function stops, the calculating of the policy is then complete and the agent will perform the optimal action, according to the calculated policy. Of course, due to the randomness of choosing belief points to upgrade the value function, the PERSEUS algorithm is sub-optimal. However, PERSEUS makes sure that, every time it is executed, it always improves the value function if possible, making it come closer and closer to the optimal value function every time. The advantage of the random selection of its belief points is that PERSEUS is very fast, making it more applicable in real-time situations. PERSEUS has already shown its potential with POMDPs in [1].

5 Predictive State Representations

A PSR is defined as a tuple $< Q, O, A, M, m, P(Q|h_0) >$, where O and A are again the sets of possible observations and actions. In PSRs it is assumed, just like in a POMDP, that an agent takes an action $a \in A$ every timestep and it gets an observation $o \in O$ as a result of this action. Instead of maintaining a probability vector over all possible world states, PSRs use a probability vector over **core tests**, called a *prediction vector.* To explain what core tests are, first the idea of normal tests has to be explained. A normal test is a sequence of alternating action/observation pairs which can occur during a finite number of timesteps in the future. The probability assigned for test $t = a_1 o_1 \dots a_n o_n$ is the conditional probability of correctly predicting the observations from this test as a result of taking the actions of this test and a history $h : P(t|h) = P(o_1 \dots o_n | h, a_1..a_n)$. The history at $t = 0$ is called the null history, h_0. The history $h = a^1 o^1 \dots a^k o^k$ is a series of alternating actions and observations in the past, and the tests are future series. As one can see, this vector is purely represented by actions and observations, making it more tangible than a POMDP state vector which uses a probability distribution over a hidden state model, described in 2. It has been shown by Littman et al.[2] that, for every dynamical system, there exists a set of tests $Q = q_1 \dots q_n$, called the **core tests**, whose predictions are a sufficient statistic of history. In particular, they showed that for every test t

$$P(t|h) = P(Q|h)^T m_t$$

where m_t is some weight vector that depends on t, but not on h. The state representation in PSRs is thus a probability distribution over these core tests, called the prediction vector. The general idea is that this prediction vector can uniquely represent every different 'state'. From position x the probability of observing observation o_1 after performing action a_1 can be much higher as observing o_2, whereas in another position or 'state' y, the probabilities may be the other way around. The prediction vector is also updated every timestep, as a result of the action a and observation o, according to the formula:

$$p(Q|hao) = \left(\frac{p^T(Q|h)M_{ao}}{p^T(Q|h)m_{ao}} \right) \tag{7}$$

where M_{ao} and m_{ao} are matrices with the model parameters. M_{ao} is of size $|Q| \times |A|$ for each combination a, o and m_{ao} is of size $|Q| \times 1$ for each a, o-combination. These matrices should be supplied beforehand and define how the prediction vector is updated according to this formula. Because the prediction vector $p(Q|hao)$ is a collection of individual probabilities, its sum does not need to be normalized to 1, in contrary to the belief vector in POMDPs. Formula 7 looks a bit like a Bayes' rule, but it is not; the matrices M_{ao} and m_{ao} can even contain negative values.

To make use of any planning and value iteration at all, rewards are needed. Rewards are modeled in PSRs as additional observations, and change the tests to $t = a_1(r_1o_1)\ldots a_n(r_no_n)$ and the history to $h = a^1(r^1o^1)\ldots a^n(r^ko^k)$. The model parameters matrix M_{ao} also gains an extra dimension; it is now three-dimensional (actions, observations and rewards) instead of two-dimensional. Because rewards are additional observations, the formula for the probability an observation is made with history h_t is of the same shape as the formula for the probability of a reward with history h_t, and according to [3] this is:

$$P(r|h_t, a) = P^T(Q|h_t) \sum_{o \in O} m_{a,(r,o)} \tag{8}$$

and

$$P(o|h_t, a) = P^T(Q|h_t) \sum_{r \in R} m_{a,(r,o)} \tag{9}$$

which leads us to the expected immediate reward $R(h_t, a)$ for action a at history h_t:

$$R(h_t, a) = \sum_{r \in R} r \cdot P(r|h_t, a) = P^T(Q|h_t) \sum_{r \in R} r \cdot \sum_{o \in O} m_{a,(r,o)} \tag{10}$$

in which the $P^T(Q|h_t)$ is the prediction vector.The latter part of 10,

$$\sum_{r\in R} r \cdot \sum_{o\in O} m_{a,(r,o)}$$

can also be computed in advance and stored in a vector n_a (for all $a \in A$), which make a reward matrix R of size $|A| \times |Q|$. This has as a consequence that rewards are a linear function of the prediction vector according to

$$R(h_t, a) = P^T(Q|h_t)n_a \tag{11}$$

and the rewards can be summed out of the matrices M_{aro} and m_{aro}: $M_{ao} = \sum_{r\in R} M_{aro}$ and $m_{ao} = \sum_{r\in R} m_{aro}$. That also means that the formula for updating the prediction vector does not need to be changed, because it can still use the M_{ao} matrix instead of the larger M_{aro} matrix and m_{ao} instead of m_{aro}. The rewards have now become a linear function of the prediction vector, implying there is a scalar reward for every core test outcome, making the expected current reward the sum of the expected reward over all core test outcomes. This extension does need the reward matrix R to be included in the PSR-defining tuple, resulting in $< Q, O, A, R, M, m, P(Q|h_0) >$, with R of size $|Q| \times |A|$, and every column a of R is the vector n_a from formula 11. Because of the fact that rewards are additional observations, increasing $|O|$, there can possibly be more core tests than there would be without rewards. This is because of the total number of possible observations changes from $|O|$ to $|O| \times |R|$. This does not mean that the number of core tests will increase with the same magnitude; it will at least stay equal, but could increase due to more diversity in tests.

6 Value iteration on PSRs

Since reward is now modeled for one timestep, the future discounted reward and a value function can be defined. The value function for timestep t, assuming action a will be performed after history h, is given by the Bellman equation:

$$V(h_t a) = R(h_t, a) + \gamma \sum_{o \in O} P(o|a, h_t) V(h_t ao) \tag{12}$$

To find the optimal action to perform, formula 12 should be maximized for action a, according to

$$V(h_t) = \max_a V(h_t a) \tag{13}$$

in which $V(h_t a)$ can be rewritten using formulae 7 – 10. As in POMDPs, we can easily show by induction that the value function in PSRs can be expressed as the upper surface of a set of vectors, i.e.

$$V(h) = \max_{\alpha_i} P(Q|h)^T \alpha_i$$

where α_i is a vector of the value function. If we set $\alpha_{h_t} = \arg\max_{\alpha_i} P(Q|h_t ao)^T \cdot \alpha_i$, then the Bellman equation (12) reads:

$$\begin{aligned} V(h_t a) &= P(Q|h_t)^T n_a + \gamma \sum_{o \in O} \left(P(Q|h_t)^T m_{ao}\right) \left(P(Q|h_t ao)^T \alpha_{h_t}\right) \\ &= P(Q|h_t)^T n_a + \gamma \sum_{o \in O} (P(Q|h_t)^T m_{ao} \left(\frac{P(Q|h_t)^T M_{ao}}{P(Q|h_t)^T m_{ao}}\right) \alpha_{h_t} \\ &= P(Q|h_t)^T n_a + \gamma \sum_{o \in O} P(Q|h_t)^T M_{ao} \alpha_{h_t} \\ &= P(Q|h_t)^T \left(n_a + \gamma \sum_{o \in O} M_{a,o} \alpha_{h_t}\right) \end{aligned} \tag{14}$$

Thus, the value of $V(h)$, through formulae 13 and 14, is calculated by back-projecting all vectors in the current horizon value function one step from the future and returning the inner product of the current prediction vector and the vector which maximizes this value.

As already stated earlier in this paper, the value function for PSRs is also PWLC. Due to this shape, there also is an incremental pruning algorithm for PSRs, called PSR-IP, described in [3]. This is an adaptation of the POMDP-IP algorithm from [5] such that it fits PSRs, and still retains the PWLC value function shape. PSR-IP uses the same intermediate W- and V-sets as POMDP-IP, but to make it compatible with PSRs, there are two things necessary. First of all, the rewards need to be a linear function of the prediction vector. This has been calculated in advance, as can be seen in formula 11. Second, the value function needs to remain PWLC. This is the case, because the value function is defined by the upper surface of the expected rewards functions for all tests, and is a piecewise linear function over PSR prediction vectors. PSR-IP should now technically be applicable for value iteration, but there still is one problem. According to [3] it could very well be the case that, because there is not a simple definition of when a prediction vector is valid, some vectors are added to the value function which are not valid. This does not invalidate the PSR-IP algorithm, but it does however have a negative effect on the efficiency, which may even be severe. This effect can be reduced by adding more constraints to the purge routine. the efficient use of constraints is the key to successfully performing incremental pruning. Nevertheless, PSR-IP still takes relatively much time to complete, like POMDP-IP or many other incremental pruning algorithms; another option is to use Q-learning with overlapping CMAC grids ([8]) for these problems. Q-learning tries to find the optimal policy by taking random actions with a probability of μ, which starts from a value close to 1 and converges to zero during execution. It stores the outcome of the random action, an exploration action, and updates its policy with it. To find the optimal policy eventually, however, the agent must try out each action in every state many times. Q-learning does guarantee that the optimal policy is found, but it may simply take a lot of time for all the actions to be performed a sufficient number of times to extract an optimal policy from it. Unfortunately there are no execution times available in [3], but a number of iterations performed with the Q-learning algorithm is shown; These range from 10,000 to 200.000 to get the same results as PERSEUS$_{PSR}$ gets in about 150 iterations. In Matlab, simple counting from 1 to 200,000 in a while-loop already takes 0.6 seconds on average (on a 2.4 GHz); performing 200,000 iterations of Q-learning will take much more time. For execution times of PERSEUS$_{PSR}$ and the relation to these 0.6 seconds see section 9.

7 PERSEUS$_{PSR}$: Randomized point-based value iteration on PSRs

There is only little change in the workings of PERSEUS needed to make it suited for PSRs. The overall scheme is quite identical, as can be seen in the main scheme of PERSEUS$_{PSR}$ below, but the **backup**-operator should work on randomly chosen prediction vectors instead of randomly chosen belief points (which are probability distributions over world states). For analogy with PERSEUS this set of sampled prediction vectors will also be called B. Furthermore, the sampled prediction vectors need to be sufficiently different. Sampling many equal vectors has as an effect that the B-set is not diverse enough, making it a bad representative of the world, which in its turn leads to poor policies.

The rewards in a PSR are modeled as additional observations, but that does not make much of a difference in the overall workings of PERSEUS$_{PSR}$, since the reward received by the system is a linear combination of the prediction vector, and can be written as a function of the action a and the history h according to equation 11. These two requirements were also needed for the PSR-IP algorithm, as already mentioned. The backup-operator can be directly computed from equation 14:

$$\begin{aligned}
\alpha_{n+1}^{b_t} &= backup(b_t) \\
&= \arg\max_a \left(R(h_t, a) + \gamma \sum_{o \in O} P(o|a, h_t) \arg\max_i P(Q|h_t a o) \alpha_n^i \right) \\
&= \arg\max_a P(Q|h_t, a) \left(n_a + \gamma \sum_{o \in O} M_{a,o} \alpha_{h_t} \right) \qquad (15)
\end{aligned}$$

with $\alpha_{n+1}^{b_t}$ the vector (with corresponding action) which maximizes the reward of performing a certain action in timestep n+1, from the point of view where prediction vector b with history h_t is considered. One other obvious but important modification is the use of formula 7 instead of formula 1, since we are dealing with core tests instead of states here. That makes the main scheme of PERSEUS$_{PSR}$ look like this:

1. Randomly collect a set B of reachable prediction vectors (probability distributions over core tests) and initialize the value function V_0 as one vector with all its components minimal, to $\frac{1}{1-\gamma} min_{b,a} r(b,a)$. Starting with V_0, PERSEUS$_{PSR}$ will perform value function updates in step 3-5 until some convergence criterion is met.

2. Set $V_{n+1} = \emptyset$. Initialize $\tilde{B}$ to B, where $\tilde{B}$ is the set of non-improved prediction vectors.

3. Sample a prediction vector b uniformly at random from $\tilde{B}$, and compute $\alpha = backup(b)$.

4. If $b \cdot \alpha \geq V_n(b)$ then add α to V_{n+1}, otherwise add $\alpha' = argmax_{\alpha^i \in V_n} b \cdot \alpha^i$ to V_{n+1}.

5. Compute $\tilde{B} = \{b \in \tilde{B} : V_{n+1}(b) - \epsilon < V_n(b)\}$ and go to step 2 if $\tilde{B}$ is not empty yet.

It is clear that the two varieties of PERSEUS are very much alike. The reason for this is that the focus of PERSEUS is the PWLC value function, which is present in POMDPs as well as PSRs. Next to these very important changes, there have been a few other minor modifications, due to the fact that observations in PSRs are (a,o)-combinations. These small changes will not be discussed as they have no impact on the overall workings of PERSEUS$_{PSR}$. To evaluate PERSEUS$_{PSR}$, it needs to be run on some problems, which are transformed from a POMDP setting to a PSR setting. The next section will cover this.

8 Transforming a POMDP to a PSR

PERSEUS$_{PSR}$ has been tested on several of the same problems as James, Singh & Littman have tested their incremental pruning and Q-learning algorithms on in [3]. These problems are the 4×3, Cheese, Paint, Shuttle, and Tiger problems, and are obtained from the webpage of T. Cassandra, [4]. An added problem, also from [4], is the bridge problem, which has a large amount of observations. These problems originate in the POMDP setting, and have been transformed to a PSR setting using the algorithm described in [2]. In this algorithm there is a notion of independence; test *aot* must be independent of S. A test t is independent of a set of tests S if its *outcome vector* is linearly independent of the outcome vectors of the tests in Q_{temp}. The outcome vector for test *aot* is recursively defined as $u(aot) = (T^a O^{ao} u(t)^T)^T$ and $u(\varepsilon) = \phi$ with ε the null test and ϕ a $1 \times n$-vector of all 1s. This makes the conversion algorithm as follows:

```
$Q \leftarrow$ search($null\_test, \{\}$)
search($t, Q_{temp}$):
    for each $a \in A, o \in O$
        if $aot$ is linearly independent of $Q_{temp}$
            then $S \leftarrow$ search($aot, Q_{temp} \cup \{aot\}$)
        return S
```

This algorithm performs a depth-first search through the entire action and observation space, making a one-step extension of its tests each time, starting with the null test. Every extension will be checked for linear independence (which can be done in a variety of ways) of the already found tests and the algorithm halts when it can find no more new independent tests. The outcome will be the entire set of **core tests**, and then what remains is to calculate the weights matrix W(which is not used in PERSEUS$_{PSR}$), and the model parameters M_{ao} and m_{ao}. This is done by first creating matrix U, a matrix formed by concatenating the outcome vectors for all core tests $\in Q$. Due to the linear independency between all elements of Q_{temp}, the columns of $T^a O^{ao} U$ are linearly dependent on the columns of U for all combinations of a and o. This enables the construction of the $|Q| \times |Q|$ sized weights matrix W according to $T^a O^{ao} U = U W^T$. The matrix M_{ao} is also of size $|Q| \times |Q|$ for each a, o combination and is defined as $M_{ao} = (U^+ T^a O^{ao} U)^T$ and m_{ao} is a $1 \times q$ vector formed by $m_{ao} = (U^+ T^a O^{ao} \phi^T) T$ in which U^+ is the pseudoinverse of U. As one can see, this program makes a form of depth-first search, making it a costly but necessary procedure for converting POMDPs to PSRs.

The problems converted by this algorithm are listed in table 1 and 3, where one can also compare them to their POMDP variety with regard to the problem size. Striking is, that most of the problems have the same amount of PSR core tests as POMDP states. The size of Q_{temp} is bounded in cardinality by $|S|$, the number of states in the corresponding POMDP, due to its linear independence. Therefore, no core test in Q is longer than $|S|$ action-observation pairs. The similarity in size is also visible in tables 1 and 3, where the number of states is mostly equal to the number of core tests, even for the large problems.

One more thing has to be noted about the bridge problem. The rewards in the specification of the bridge problem were set as 'costs', instead of 'rewards', which needed the rewards to be multiplied by -1. Without this extra step no heads or tails could be made of the computed policy. The convergence was there, but not smooth at all, the rewards (for both manners) were fluctuating heavily with relatively large standard deviations and apparently not converging and the number of vectors did not approach the value found with the altered rewards. See the next section for graphs and numbers.

There has been a little speculation about the size and power of PSRs related to their POMDP varieties. First, as mentioned earlier, a PSR problem will be equal-sized or smaller than the POMDP problem due to the linear independence of core tests. Second, PSR size can also be related to the diversity of the environment, expressed by some 'diversity measure'. The size of the minimal PSR is then bounded above by the diversity of the environment, and this diversity is equal-sized or (possibly exponentially) smaller than the state set of a POMDP. Last, as a vector of *state variables*, capable of taking on diverse values, a PSR may be more powerful than the distribution over discrete states of a POMDP, according to [2].

9 Results

This section shows the results obtained with running PERSEUS$_{PSR}$ on the problems converted by the algorithm described in the previous section. It turns out that converging in rewards is not the same as being able to find vectors which improve the value function. Once a converged policy is found, the **backup**-operator still finds improved vectors for the value function, but they either do not invoke a change in the policy or they invoke a very small, unnoticeable one. The number of iterations needed for each problem to converge in policy is thus not the same number of iterations to converge in rewards. This is clearly visible if one looks at figure 3, which shows the convergence and the number of vectors which make up V for the smaller problems. The convergence graphs show the sum of the maximum of the inner products for all $b \in B$ with the vectors of the value function, i.e.

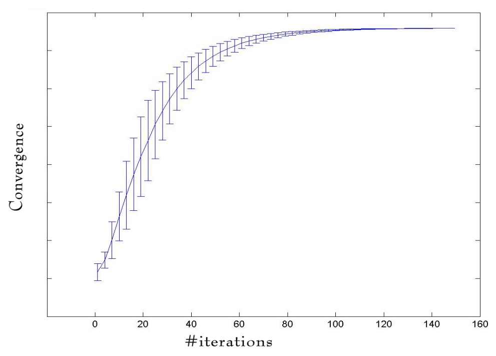
$$\sum_{b \in B} \max_{\alpha} \alpha \cdot b$$

The exact numbers are not important here, but what is is that it shows asymptotic behavior - this shows that the value function converges. A comparison with figure 5 makes the convergence even more clear. It seems as sampling more prediction vectors for B can result in an increase of $|V|$ as well. The size of $|V|$ is limited by the number of prediction vectors sampled; when one only uses 2 sampled prediction vectors then $|V|$ can of course never exceed 2. But when two different sets of 100 prediction vectors are sampled for these small problems, this could lead to a small change in the number of $|V|$ as well - as an example, the 'paint'-problem could have an average of 5 vectors for $|V|$ but with another set of sampled prediction vectors this might be 6. This explains the relatively large standard deviations for some problems in the number of vectors which make up the computed policy. For the smaller problems, sampling max. 100 prediction vectors, for the set B is enough. The bridge problem was run with 250 sampled prediction vectors, and the 4×3 problem was run with 350 prediction vectors. Stabilization of the number of vectors which make up the converged policy for these problems is visible in figure 4.

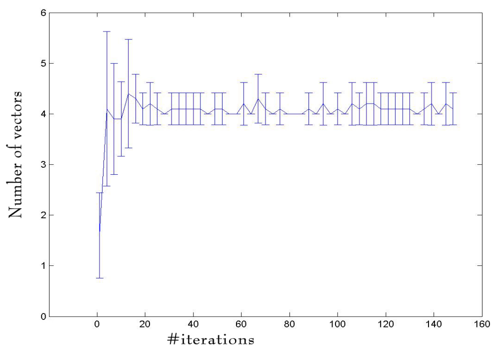
Some problems do not even have 100 sufficiently distinct prediction vectors to sample, and for these problems the maximum was used. The amount of difference these prediction vectors should have is set to 2% of the number of sampled core tests, $1/50/|B|$.

Figure 5 shows the number of iterations of PERSEUS$_{PSR}$ against the mean reward. This value has been calculated in two ways; The left column of images shows the same method as in [3]: PERSEUS$_{PSR}$ runs for several iterations, and after that the discovered policy is used to perform actions for a number of timesteps. The reward received during these timesteps is summed and divided by the number of timesteps the program ran, resulting in the average reward per timestep. The right column shows the discounted reward. The discounted reward is calculated by discounting (with $\gamma^t = 0.95^t$) and subsequently adding the rewards obtained until t timesteps have been taken that γ^t reaches $1e-5$ (which results in approximately 224 timesteps). The PERSEUS$_{PSR}$ algorithm is exactly the same, as is the computed policy; it is merely evaluated in two different ways. In the left column of figure 5 the horizontal axis is the number of iterations PERSEUS$_{PSR}$ made and the vertical axis is the average reward per timestep; in the right column of figure 5 the discounted reward can be seen. For every iteration PERSEUS$_{PSR}$ was run multiple times, constructing multiple different policies. The values in the graphs are means and standard deviations in average reward per timestep (left) and average policy reward (right) of these policies.

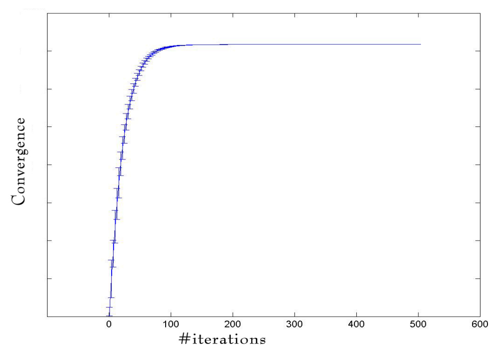
In table 1 one can see the average time it took PERSEUS$_{PSR}$ to complete, i.e. to find a policy. The times in table 1 are in seconds, and are much lower than the PSR-Incremental Pruning or the Q-learning algorithm discussed in [3], where some problems did not even complete after 8 hours. The times for Time$_{150iter}$ were produced using a single 2.4 GHz Pentium IV(no HyperThreading) processor system and are an average of running PERSEUS$_{PSR}$ 20 times for 150 iterations, whether the policy has converged or not. Most problems have already been solved long before the 150 iterations, however - 40 iterations is usually enough. The times for finding the converged policy are listed in Time$_{converged}$, the number of iterations needed can be viewed in figure 5 and 6. This way one can also get a little insight of the influence of the problem size on the execution time. Perhaps the reader remembers from the end of section 6, that counting from 1 to 200,000 in a while loop takes



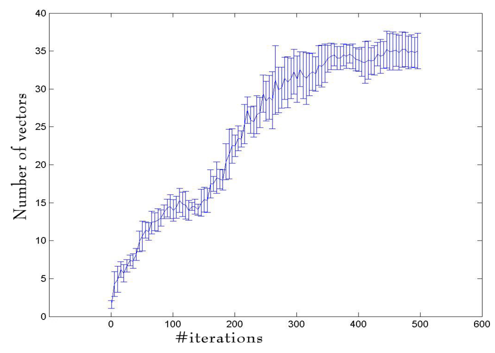
(a) paint



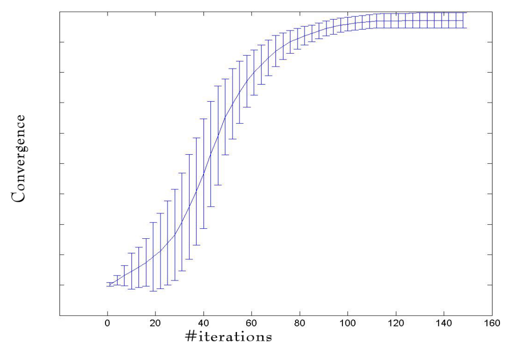
(b) paint



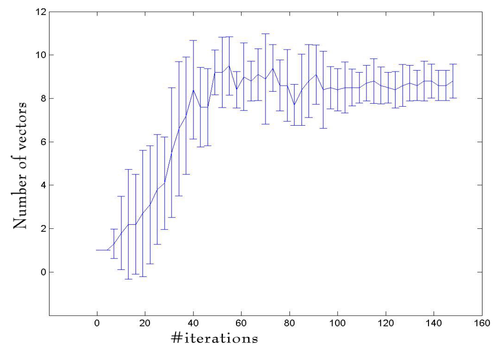
(c) shuttle



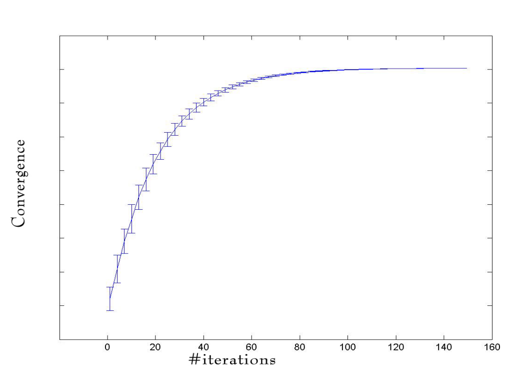
(d) shuttle



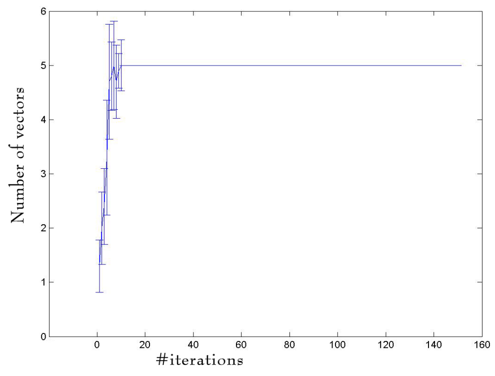
(e) cheese



(f) cheese

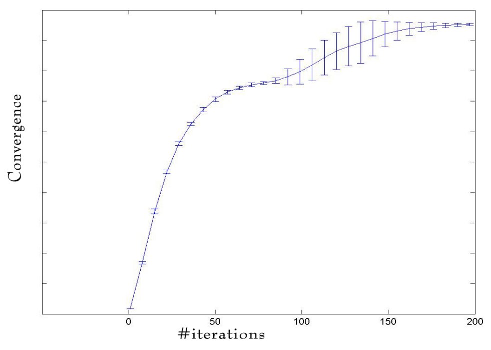


(g) tiger

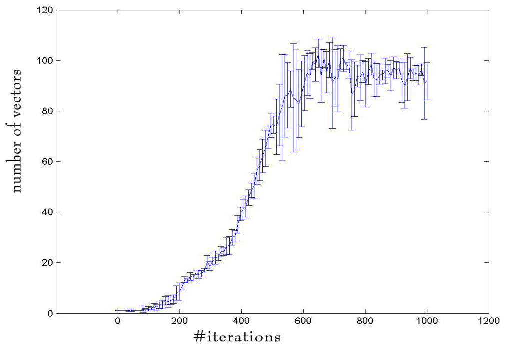


(h) tiger

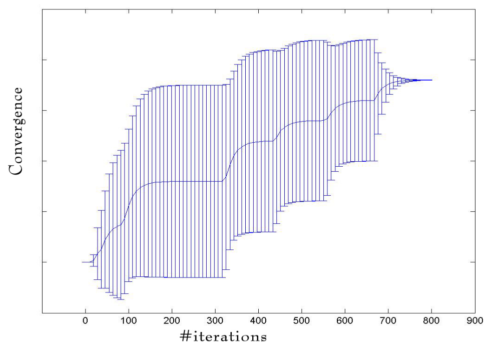
Figure 3: convergence of PERSEUS$_{PSR}$(left column) and the number of Vectors which make up V (right column).



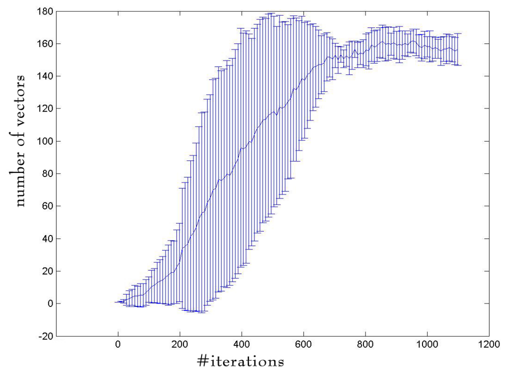
(a) bridge



(b) bridge

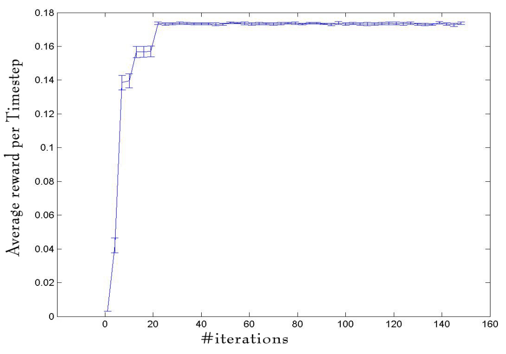


(c) 4×3

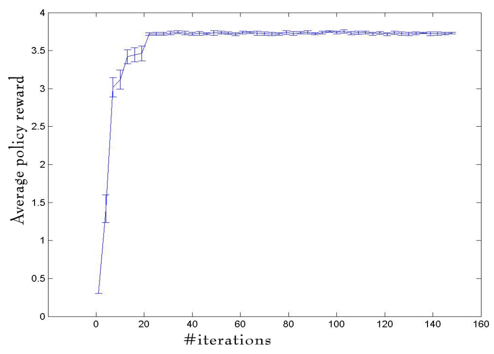


(d) 4×3

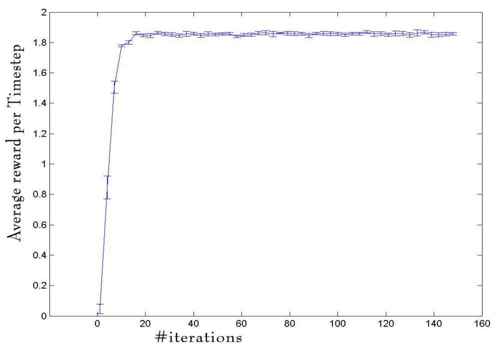
Figure 4: convergence of PERSEUS$_{PSR}$(left column) and the number of Vectors which make up V (right column).



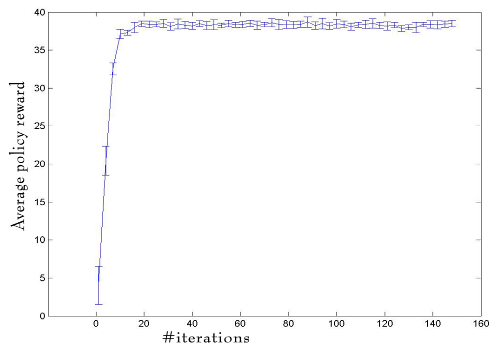
(a) paint



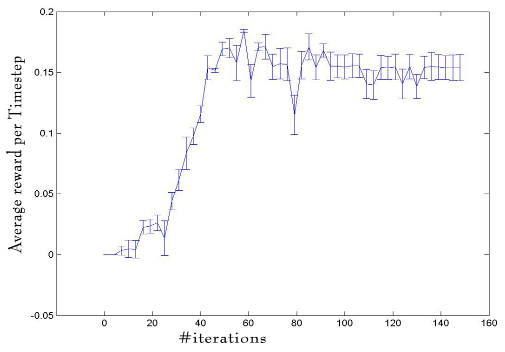
(b) paint



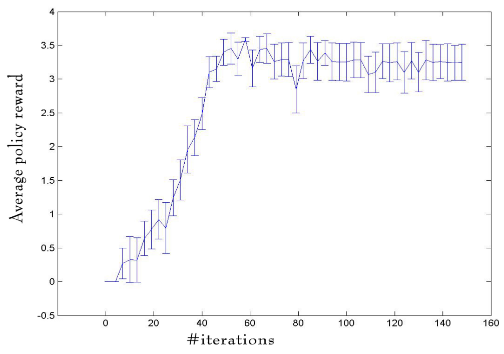
(c) shuttle



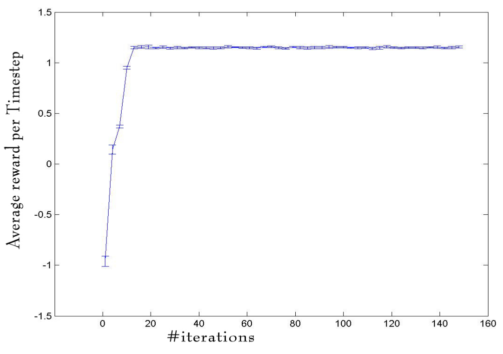
(d) shuttle



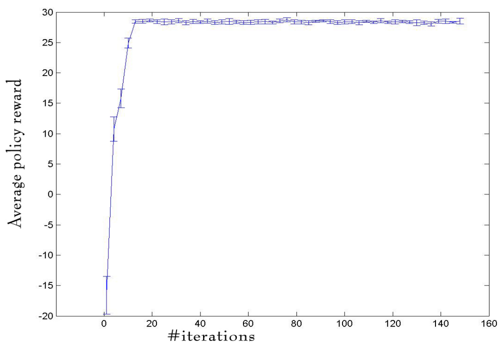
(e) cheese



(f) cheese

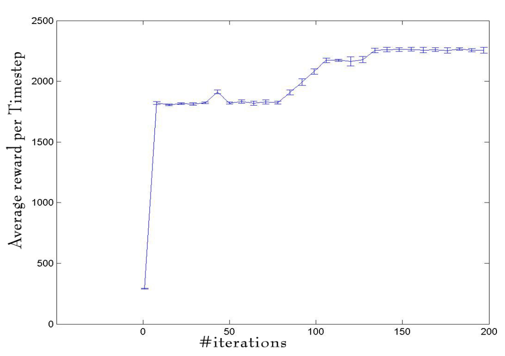


(g) tiger

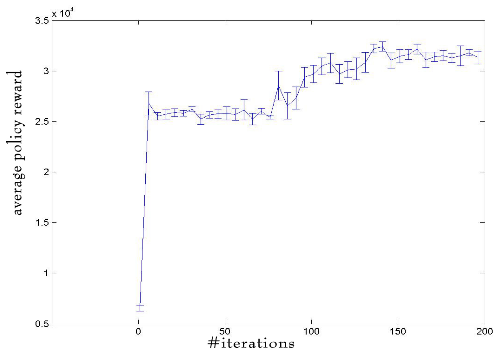


(h) tiger

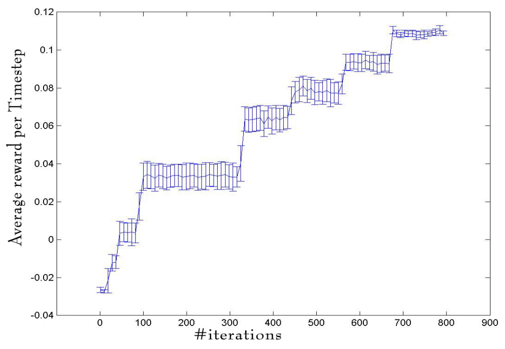
Figure 5: iterations vs. average reward per timestep (left column) and average discounted reward (right column).



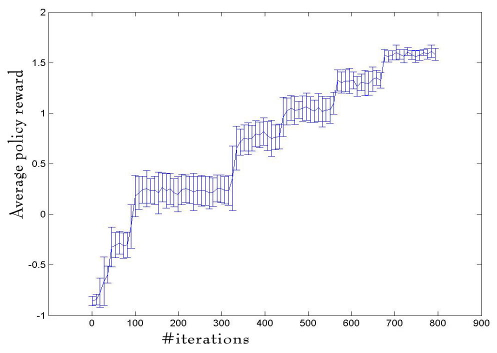
(a) bridge



(b) bridge



(c) 4×3



(d) 4×3

Figure 6: iterations vs. average reward per timestep(left column) and average discounted reward(right column).

about 0.6 seconds on average; as one can see PERSEUS$_{PSR}$ can make 150 iterations for the paint problem in that time, which is well over the number of iterations needed to construct the converged policy. From table 1 one can also see the linearity in speed of PERSEUS$_{PSR}$

Problem	POMDP $\lvert S\rvert$	PSR $\lvert Q\rvert$	$\lvert O\rvert$	$\lvert A\rvert$	Time$_{150iter}$(sec)	Time$_{converged}(sec)$
Paint	4	4	4	4	0.51	0.18
Bridge	5	5	137	12	18.2	14.6
Cheese	11	11	7	4	1.24	0.34
Tiger	2	2	6	3	0.51	0.09
4x3	11	11	14	4	4.30	0.64
Shuttle	8	7	8	3	1.59	0.16

Table 1: Execution times for PERSEUS$_{PSR}$ on small problems. Note that the observations are pairs of (o, r); therefore rewards are not mentioned separately here.

as the problems get larger; the bridge problem is of size $12 \cdot 137 \cdot 5$ and the paint problem is of size $4 \cdot 4 \cdot 4$. The linearity is clearly visible if one looks at the size of the problems and their execution times:

$$\frac{bridge}{paint} = \frac{12 \cdot 137 \cdot 5}{4 \cdot 4 \cdot 4} \approx \frac{18.1677}{0.5063} \quad \& \quad \frac{4{\times}3}{paint} = \frac{11 \cdot 14 \cdot 4}{4 \cdot 4 \cdot 4} \approx \frac{4.3089}{0.5063}$$

The other problems in table 1 have not been tested with 100 randomly sampled prediction vectors, since there were no 100 sufficiently different prediction vectors available. The number of sampled prediction vectors B has a significant effect on speed of the algorithm, therefore the execution times listed in the table cannot be used to verify linearity in speed. One could remove the constraint in the first step of PERSEUS$_{PSR}$, that the sampled prediction vectors need to be sufficiently different, or one could set the threshold to a lower value. Then, for all problems one can find $|B| = 100$. In table 2 one can see the execution times for the smaller problems when they all are run with 100 randomly sampled prediction vectors.

The execution times do not show the linearity very clearly, because it can very well be that, of the 100 randomly sampled prediction vectors, a lot are very close to each other due to the lack of the difference constraint, so one update of a vector may indeed remove a large

Problem	PSR $\|Q\|$	$\|O\|$	$\|A\|$	Time$_{150iter}$(sec)	Time$_{converged}(sec)$
Paint	4	4	4	0.51	0.18
Bridge	5	137	12	18.2	13.5
Cheese	11	7	4	1.48	0.25
Tiger	2	6	3	0.94	0.12
4x3	11	14	4	4.31	0.62
Shuttle	7	8	3	1.91	0.16

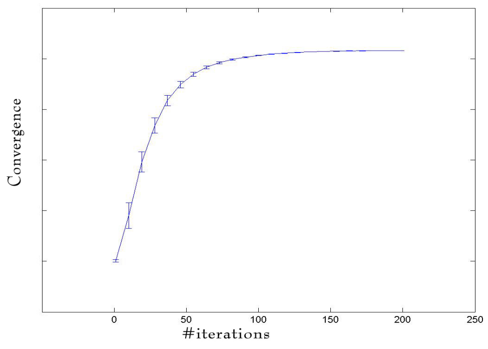
Table 2: Execution times for PERSEUS$_{PSR}$ on small problems, all with 100 randomly sampled prediction vectors.

part of $\tilde{B}$, resulting in less vectors of V to update then when one would use significantly distinct prediction vectors. This results in a faster run, distorting the view of linearity. It is however visible that no increase in execution time exceeds the amount of increase in problem size, i.e. when the problem size doubles, the execution time is double at most.

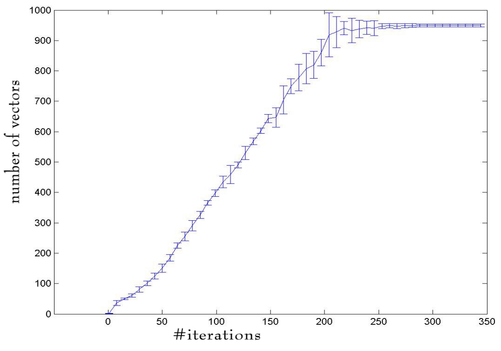
The algorithm to convert a POMDP problem to a PSR problem has also been used to transform other, well-known problems such as tigerGrid or Hallway. These problems are a lot larger and are shown in table 3 below. Again, the problems are shown with the time needed for PERSEUS$_{PSR}$ to complete, achieved on the same 2.4 GHz PIV system. For these measurements, the times needed to converge in policy were measured and averaged as well as the time to perform 200 iterations, in Time$_{200iter}$. These times were achieved using $|\tilde{B}| = 1000$, which lead to an average of 10 runs instead of 20. A sidenote to these execution times is that performing 200 iterations, as has been done in the figures, is not necessary to find the converged policy; much less iterations are needed for this. The tigerGrid times are measured for 40 iterations, hallway for 50 iterations and hallway2 for 50 iterations. The linearity in speed of PERSEUS$_{PSR}$ is, unfortunately, less clearly visible in these execution times.

Problem	POMDP $\|S\|$	PSR $\|Q\|$	$\|O\|$	$\|A\|$	Time$_{200iter}$	Time$_{optimal}$(sec)
Tiger-grid	33	33	33	5	2513	25.6
Hallway	57	57	21	5	1648	36.9
Hallway2	89	89	17	5	6572	149.4

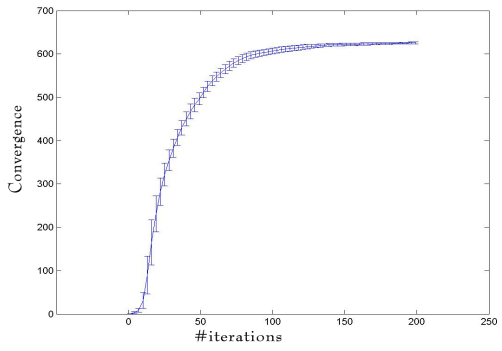
Table 3: Execution times for PERSEUS$_{PSR}$ for well-known, large POMDP problems.



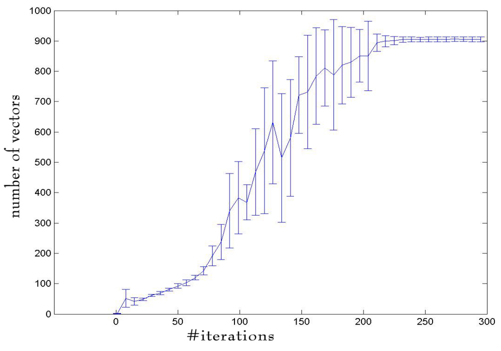
(a) tigerGrid



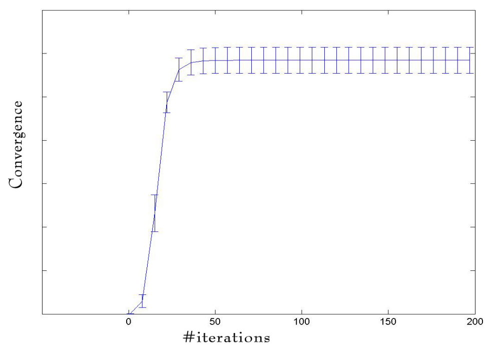
(b) tigerGrid



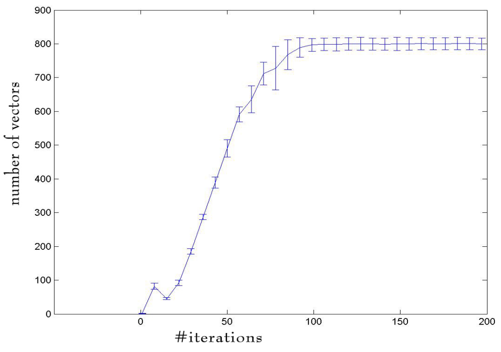
(c) Hallway



(d) Hallway

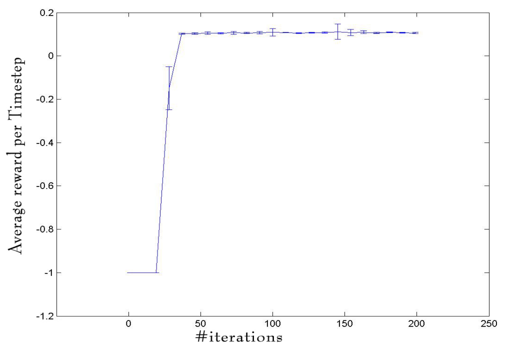


(e) Hallway2

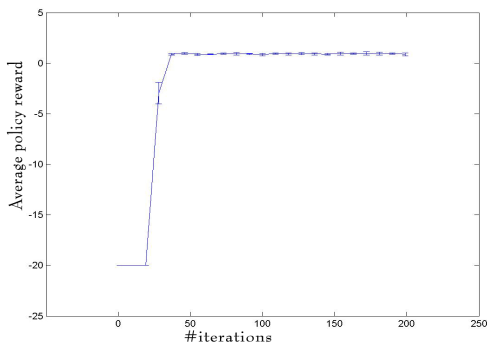


(f) Hallway2

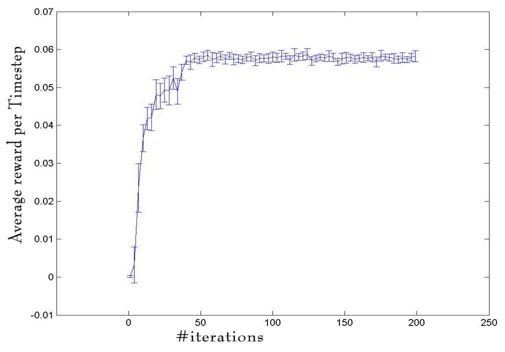
Figure 7: convergence of PERSEUS$_{PSR}$(left column) and the number of Vectors which make up V (right column). Note that the number of sampled prediction vectors is 1000.



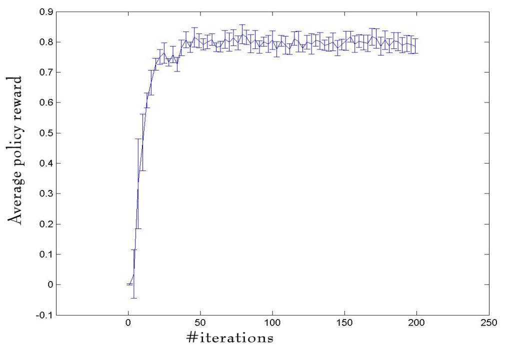
(a) tigerGrid



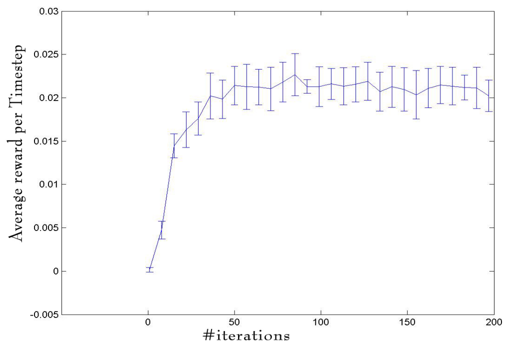
(b) tigerGrid



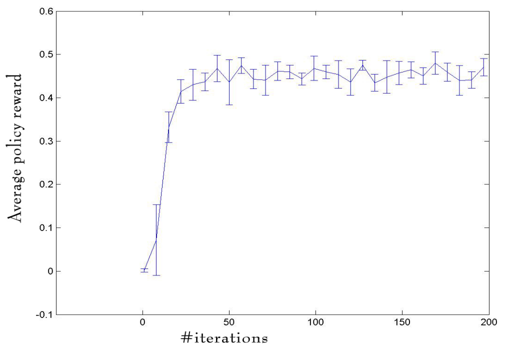
(c) Hallway



(d) Hallway



(e) Hallway2



(f) Hallway2

Figure 8: iterations vs. average reward per timestep (left column) and average discounted reward(right column).

The number of prediction vectors sampled for the set B was larger, being 1000 instead of max. 100 for the smaller problems, due to the size of the problems. There is much more 'diversity' (see section 8) in the larger problems, which makes the choice for a larger B justifiable. As said before, this can also account for the large number of vectors which make up the computed policy. In the 200 iterations of table3 the number of vectors for tigerGrid and hallway have not converged yet, but figure 7 shows at what point the converged policy is found and the number of vectors stabilizes. The reward for the entire policy and the average reward per timestep were calculated in the same way as the smaller problems. The results of these runs are shown in figures 7 and 8, showing rewards, convergence and the number of vectors which make up the computed policy. The converged policies for these large problems are yet found after only a small number of iterations, even though the problems are much larger. TigerGrid, Hallway and Hallway2 have been tested differently in [1]; when the goal state was reached, the state of the agent was reset, which did not happen in the PSR experiments in this thesis. Therefore a comparison with PERSEUS is unfortunately not possible, but it is assumed that the value PERSEUS$_{PSR}$ converges to is the maximum, like it does with the smaller problems. The exact values for average reward per timestep and average policy reward of the converted policy, computed by PERSEUS$_{PSR}$, are shown for each problem in table 4 below.

Problem	Avg. reward per Timestep	Avg. Policy reward
Tiger-grid	0.1073	0.9433
Hallway	0.0578	0.7969
Hallway2	0.0212	0.4577
Paint	0.1735	3.7327
Bridge	2261.5	31593
Cheese	0.1520	3.2329
Tiger	1.1496	28.3933
4x3	0.1085	1.5839
Shuttle	1.8562	38.3375

Table 4: average values of the converged policy for all the problems.

10 Conclusion

In this thesis it is shown that PERSEUS$_{PSR}$ is very similar to the original PERSEUS, and achieves equally good results in terms of optimal policies and speed. This makes PERSEUS$_{PSR}$ a suitable algorithm for real-time planning using PSRs. PERSEUS$_{PSR}$ does also achieve the same rewards as a result of its calculated policies as the exact PSR-IP algorithm from James & Singh. the only difference is that PERSEUS$_{PSR}$ is several orders of magnitude faster.

The speed of PERSEUS$_{PSR}$ is, like the original PERSEUS also linear in the size of the problems, which makes the algorithm very suitable for scaling purposes. It has been observed that, when sampling more prediction vectors, $|V|$ also increases; this has a negative effect on the speed of PERSEUS$_{PSR}$. Most likely, one does not need to sample as much prediction vectors for the different problems as has been done in these experiments.

Due to the fact that PSRs are a very new and powerful representation for world domains, there is quite some future work to be done in PSRs. For instance, learning the T_a and T_o matrices in a POMDP environment is currently a research topic; the same could be applied to PSRs: learning the model parameters M_{ao} and m_{ao} by interacting with the environment ([9]).

11 Bibliography

References

[1] M.T.J. Spaan and N. Vlassis, *Perseus: Randomized Point-based Value Iteration for POMDPs*, Journal of Artificial Intelligence Research, 24:195220, 2005.

[2] M.L. Littman, R.S. Sutton and S. Singh, *Predictive Representations of State*, In *Advances In Neural Information Processing Systems 14*, 2001.

[3] M.R. James, S. Singh and M.L. Littman, *Planning with Predictive State Representations*, In Proceedings of the International Conference on Machine Learning and Applications (ICMLA), 2004.

[4] A. Cassandra, *Tony's POMDP page*, http://www.cs.brown.edu/research/ai/pomdp/.

[5] A. Cassandra, M.L. Littman and N.L. Zhang, *Incremental Pruning: a simple, fast, exact method for partially observable Markov decision processes*, proceedings of the 13th annual conference on uncertainty in Artificial intelligence(UAI-97), 1997.

[6] M. Littman, A. Cassandra and Kaelbling, *Efficient dynamic programming updates in partially observable Markov decision processes*, Technical report CS-95-19, Brown University, Providence, RI, 1996.

[7] S. Singh, M.R. James and M.R. Rudary, *Predictive State Representations: A new theory for modeling dynamical systems*, University of Michigan, 2004.

[8] J. S. Albus, *A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller (CMAC)*, J. Dyn. Syst. Meas. Control, Trans. ASME 97, 220-227 (1975).

[9] M.R. James and S. Singh, *Learning and Discovery of Predictive State Representations in Dynamical Systems with Reset*, in Proceedings of the 21st International Conference on Machine Learning, Banff, Canada, 2004.

12 Acknowledgements

All these years which have eventually lead to this thesis have been a great time. I have got to know many people at the University of Amsterdam and beyond. This section is to show a little gratitude to all those people. First of all, the three most important people in my life. Wing Kiu Siu, my girlfriend for 3.5 years now, who is graduating at this moment for her (in my opinion impressive) chemistry/physics master, condensed matter sciences, also at the UvA. She has been able to stimulate and comfort me when needed, always shown interest, understanding and often a good source of fun. Jan-Pieter Schalk and Jan vd Lans are very close friends as well, and have been so for many many years. They provided good times of laughter and relaxation as well as stimulation. They have always been very reliable friends. A special thanks to these three people. At the University of Amsterdam, I started in the year 2000. Soon I got to know Niels ”Niel2” Monshouwer, Ruben ”G00se”/”0|2i0nXXL” Boumans, Niels ”N1els”/”Arnie” Roossen, Aron Abbo, Michiel Kamermans, Martijn Stegeman, Edwin vd Thiel,Paul Ruinard, Daan Vreeswijk, Joram Rafalowicz, Janneke Zwaan, Jochel Liem, my bandpeople Peter Hauwert, Peter Deuss, Joris Kreb, Rutger van Oel, Bart Luttikhuis, Karel de Jongh, Janneke V and Rene (the gatekeeper at my faculty, who has a lot of wisdom about everyday life), and many others. Together with several of them I participated in the governing board of the study association for computer sciences of Amsterdam (Via) which was a very instructive and fun experience. I have been active at the Via for several years after that as well, organizing events etc. Lately, mainly due to this thesis and the graduation accompanied with it, I have been less active there but I have had a lot of contact with my supervisor, Nikos Vlassis, who I think is a very nice and intelligent man. He gave me some useful tips about how to ”do” a thesis and he always knew how to stimulate me even more. During the making of this thesis I have also come into contact with Michael R. James, from the university of Michigan. He helped me a lot with providing the PSR problems and answering questions. Thanks, Michael! I'm going to buy you a drink when we meet, someday. And last but certainly not least, My parents: Bep & Ton Wessling. Always supportive and highly interested, even though they're terrible

with computers & computer science.. Amazing, though, how much they still understand of my thesis..;-) Thanks a lot for a good and fruitful childhood, with much support and love. Hopefully I have mentioned everyone and there are no people who feel forgotten now; If so, sue me.
Ton Wessling