Prioritizing Point-Based POMDP Solvers

Guy Shani and Ronen I. Brafman and Solomon E. Shimony
Computer Science Department, Ben Gurion University, Israel
{shanigu,brafman,shimony}@cs.bgu.ac.il

***Abstract*—Recent scaling up of POMDP solvers towards realistic applications is largely due to point-based methods that quickly converge to an approximate solution for medium-sized problems. These algorithms compute a value function for a finite reachable set of belief points, using *backup* operations. Point based algorithms differ on the selection of the set of belief points, and on the order by which backup operations are executed on the selected belief points.**

We first show how current algorithms execute a large number of backups that can be removed, without reducing the quality of the value function. We demonstrate that the ordering of backup operations on a predefined set of belief points is important.

In the simpler domain of MDP solvers, prioritizing the order of equivalent backup operations on states is known to speed up convergence. We generalize the notion of prioritized backups to the POMDP framework, showing how existing algorithms can be improved by prioritizing backups. We also present a new algorithm, Prioritized Value Iteration (PVI), and show empirically that it outperforms current point-based algorithms.

Finally, a new empirical evaluation measure (in addition to the standard runtime comparison), based on the number of atomic operations and the number of belief points, is proposed, in order to provide more accurate benchmark comparisons.

I. Introduction

Many interesting reinforcement learning (RL) problems can be modeled as partially observable Markov decision problems (POMDPs), yet POMDPs are frequently avoided due to the difficulty of computing an optimal policy. Research has therefore focused on approximate methods for computing a policy (see e.g. Brafman (1997), Poupart and Boutilier (2004), Pineau et al. (2003)). A standard way to define a policy is through a value function that assigns a value to each belief state, thereby also defining a policy over the same belief space. Smallwood and Sondik (1973) show that this value function can be represented by a set of vectors and is hence piecewise linear and convex.

A promising approach for computing value functions is the point-based method. Algorithms of this family compute a value function over a finite subset of the reachable belief space. An optimal solution for a subset of the belief space may not be optimal for the entire belief space. However, point-based methods assume that the solution would generalize well to other, unobserved belief points. Generalizing to the entire belief space is possible through the use of the vector representation of a value function for POMDPs. The main contribution of this paper is a framework for accelerating point-based solvers by smartly ordering the sequence of value-function updates over a set of belief points. We also propose an improved method for finding a "good" finite set of belief points.

Improving a value function represented by vectors can be done by performing a *backup* operation over a single belief state, resulting in a new vector that can be added to the representation of the value function. Even though a vector is computed for a single belief state, it defines a value over the entire belief space, though this value may not be optimal for many belief states. A single backup operation can therefore, and in many cases does, improve the value function for numerous belief points. Backup operations are relatively expensive, and POMDP approximation algorithms can be improved by reducing the number of backup operations needed to approximate the optimal policy.

For the simpler domain of Markov decision processes (MDPs), it was previously observed (Moore & Atkeson, 1993; Bonet & Gefner, 2003; Wingate & Seppi, 2005) that the order by which backup operations are executed over states can change the convergence rate of the value function. For example, as the value for a state is influenced by the values of its successors it is more useful to execute a backup operation for a state only after values for its successors are computed. In an MDP it is easy to find the set of predecessors for a given state making backward state space traversals possible. Methods that define a backup sequence can be viewed as performing backups in an order of decreasing *priorities* for states.

We have recently introduced the idea of using prioritization in POMDPs (Shani et al., 2006), demonstrating how similar ideas can be applied in the more difficult POMDP domain. A direct implementation of the techniques used for MDPs is not possible. First, one cannot efficiently find the set of predecessors for a belief state, which may have unbounded size. Second, a backup operation for a belief state potentially improves the value of many other belief states as well, therefore affecting belief states that are not direct predecessors of the improved state.

In this extended version of the paper, we initiate a more disciplined examination of backup operation ordering. We raise the following issue: suppose that we hold everything else (such as set of belief points, etc.) constant, what would be the sequence of backups that would lead the algorithm to converge most quickly to the correct policy? This poses a meta-reasoning problem that is hard in and of itself, and certainly not one we could expect a POMDP algorithm to solve at runtime. However, we can attempt an approximate solution to this meta-reasoning post-facto, and examine the backup performance of various backup schemes in this light, and these new empirical results are illuminating.

An orthogonal question to the order of backups is the

Partially supported by the Paul Ivanier center for robotics and production management.

selection of the belief subsets. Previous methods suggested to cover the belief space as best as possible using a finite number of reachable points (Pineau et al., 2003), or to use a random selection of reachable points (Spaan & Vlassis, 2005). These methods take two extreme views — the first generates very good belief sets but requires extensive computations, while the second is extremely fast but produces random sets. We show here that in complex domains it is unlikely that a random walk will quickly reach important parts of the domain. We continue to suggest and evaluate an heuristic method that is very fast to compute and produces good belief sets.

Another important issue this paper addresses is the scheme for reporting experimental results evaluating the performance of point-based algorithms. Previous researchers have implemented their own algorithms and compared its performance to previous published results, usually reporting Average Discounted Reward (ADR) as a measure of the quality of the computed policy, and convergence time, over well-known benchmarks. While the ADR of a policy is identical, although noisy, over different implementations, the convergence time is an insufficient measurement. Execution time for an algorithm is highly sensitive to variations in machines (CPU speed, memory capacity), selected platform (operating system, programming language) and implementation efficiency. Although we comply with the commonly used result reporting scheme, additional measures are also reported, designed to help future publications to provide a fair comparison.

II. Background and Related Work

We begin with an overview of MDPs and POMDPs, the belief space MDP, and how a solution to a POMDP is computed. We then provide an short introduction to point-based methods for solving POMDPs, and explain how prioritization has been used before in the context of MDPs.

A. MDPs, POMDPs and the belief-space MDP

Markov Decision Processes (MDPs) are designed to model autonomous agents, acting in a stochastic environment. Consider for example a robot traveling through a maze. The robot starts at some location and can either move forward, turn left, or turn right. As the robot moves its location may change, and thus, the environment, which includes the location of the robot, changes. The assumption is that the environment changes only as the result of the agent actions. The robot must reach some goal state, such as the exit door, or alternatively, collect rewards, such as items that are scattered through the maze.

Formally, an MDP is a tuple $\langle S, A, tr, R \rangle$ where:

- S is the set of all possible world states. In the above example, the environment state is the location and orientation of the robot.
- A is a set of actions the agent can execute. Our robot can only turn left, right, or move forward.
- $tr(s, a, s')$ defines the probability of transitioning from state s to state s' using action a. The transition function models the stochastic nature of the environment, such as the robot attempting to move forward but failing due to engine malfunction or because the wheels where slipping.
- $R(s, a)$ defines a reward the agent receives for executing action a in state s. Action costs can be modeled as negative rewards. In our example the robot receives a reward for getting out of the maze or for collecting an item. The robot may pay a cost each time it moves, modeling the energy loss incurred by the move.

An MDP models an agent acting in an environment where it can directly observe the state it is at.

Realistically, a robot does not know where it is located within a maze. It has sensors that provide observations such as nearby walls. These sensors are imperfect, meaning that they sometimes detect a wall where none exist, and sometimes the sensors fail to detect an existing wall. Now, in order to find its way through the maze the robot must also gather information about the environment state — its own location within the maze.

A Partially Observable Markov Decision Process (POMDP) is designed to model such agents that do not have direct access to the current state, but rather observe it through noisy sensors. A POMDP is a tuple $\langle S, A, tr, R, \Omega, O, b_0 \rangle$ where:

- S, A, tr, R compose an MDP, known as the underlying MDP. This MDP models the behavior of the environment.
- Ω is a set of available observations — the possible output of the sensors. In the example above the set of observations consists of all possible wall configurations.
- $O(a, s, o)$ is the probability of observing o after executing a and reaching state s, i.e. the sensor model, which incorporates the sensor noise.

As the agent is unaware of its true world state, it must maintain a *belief* over its current state — a vector b of probabilities such that $b(s)$ is the probability that the agent is at state s. Such a vector is known as a belief state or *belief point*. b_0 defines the initial belief state — the belief of the agent over the state space before it has executed or observed anything.

Given a POMDP it is possible to define the belief-space MDP — an MDP over the belief states of the POMDP. The transition from belief state b to belief state b' using action a is deterministic given an observation o and defines the τ transition function. That is, we denote $b' = \tau(b, a, o)$ where:

$$b'(s') = \frac{O(a, s', o) \sum_s b(s) tr(s, a, s')}{pr(o|b, a)} \quad (1)$$

$$pr(o|b, a) = \sum_s b(s) \sum_{s'} tr(s, a, s') O(a, s', o) \quad (2)$$

Therefore, τ is computed in $O(|S|^2)$.

B. Value Functions for POMDPs

It is well known that the value function V for the belief-space MDP can be represented as a finite collection of $|S|$-dimensional vectors known as α vectors. Thus, V is both piecewise linear and convex (Smallwood & Sondik, 1973). A policy over the belief space is defined by associating an action a to each vector α, so that $\alpha \cdot b = \sum_s \alpha(s) b(s)$ represents the

value of doing action a in belief state b and following the policy afterwards. It is therefore standard practice to compute a value function — a set V of α vectors. The policy π_V is immediately derivable using:

$$\pi_V(b) = \text{argmax}_{a:\alpha_a \in V} \alpha_a \cdot b \tag{3}$$

We can compute the value function over the belief-space MDP iteratively:

$$V_{n+1}(b) = \max_a [b \cdot r_a + \gamma \sum_o pr(o|a,b) V_n(\tau(b,a,o))] \tag{4}$$

where $r_a(s) = R(s,a)$ is a vector representation of the reward function. The computation of the next value function $V_{n+1}(b)$ out of the current V_n (Equation 4) is known as a *backup* step. The backup step can be implemented efficiently (Pineau et al., 2003; Spaan & Vlassis, 2005) by:

$$backup(b) = \text{argmax}_{g_a^b : a \in A} b \cdot g_a^b \tag{5}$$

$$g_a^b = r_a + \gamma \sum_o \text{argmax}_{g_{a,o}^\alpha : \alpha \in V} b \cdot g_{a,o}^\alpha \tag{6}$$

$$g_{a,o}^\alpha(s) = \sum_{s'} O(a,s',o) tr(s,a,s') \alpha^i(s') \tag{7}$$

Note that the $g_{a,o}^\alpha$ computation (Equation 7) does not depend on the belief state b and can therefore be cached for future backups. All the algorithms we implemented use caching to speed up backup operations. Without caching the $g_{a,o}^\alpha$ results, the backup process takes $O(|S|^2|V||\Omega||A|)$.

While it is possible to execute full backups for V over the entire belief space, hence computing an optimal policy (Cassandra et al., 1997), the operation is computationally hard. Various approximation schemes attempt to decrease the complexity of computation, potentially at the cost of optimality.

A value function can be defined using other representation, such as a direct mapping between belief states and values. Given such a representation we use the H operator, known as the Bellman update, to compute a value function update:

$$Q_V(b,a) = b \cdot r_a + \gamma \sum_o pr(o|a,b) V_n(\tau(b,a,o)) \tag{8}$$

$$HV(b) = \max_a Q_V(b,a) \tag{9}$$

The computation time of the H operator is $O(T_v|S|^2|O||A|)$, where T_v is the time it takes to compute the value of a specific belief point using the value function V.

C. Point Based Value Iteration

Computing an optimal value function over the entire belief space does not seem to be a feasible approach. A possible approximation is to compute an optimal value function over a finite subset B of the belief space (Lovejoy, 1991). Unfortunately, an optimal solution over B does not guarantee optimality over belief points not in B. It is therefore possible that for some reachable belief states (which are not included in B) the resulting value function is sub-optimal. Such a schemes are based on the (empirically verified) assumption, that the computed value function will generalize well for other belief states not included in B.

Point-based algorithms (Pineau et al., 2003; Spaan & Vlassis, 2005; Smith & Simmons, 2005) choose a subset B of the belief points that is reachable from the initial belief state through different methods, and compute a value function only over the belief points in B.

The Point Based Value Iteration (PBVI) algorithm (Pineau et al., 2003), (Algorithm 1), begins with $B = b_0$, and at each iteration computes an optimal value function for the current belief points set. After the value function has converged the belief points set is expanded with all the most distant immediate successors of the previous set. Following Pineau et al. we used the L_2 distance metric in our reported experiments[1].

Given the ever expanding belief space it is clear that at the limit, the belief set B_∞ will cover the entire reachable belief space. Thus, at the limit, PBVI will compute an optimal value function over all reachable beliefs. However, at the limit, the number of α-vectors can also be unbounded, making the point-based backup intractable.

PBVI has a number of shortcomings, not allowing it to scale up to larger domains. First, the belief expansion procedure (Algorithm 3) requires the time consuming computation of distances. Computing a distance between any two belief points requires $|S|$ operations. As we have $|B|$ belief points, and each belief point has $|A||O|$ successors, computing the expanded belief space requires $|B|^2|A||O||S|$ operations. To reduce this computation cost, Pineau et al. also suggest to randomly select a successor for each belief-action pair, reducing the computation to $|B|^2|A||S|$ at the cost of missing distant successors. The value function update phase of PBVI (Algorithm 2) requires a complete backup of all the belief points in the set B in an arbitrary order. Such a backup sequence is time consuming and as we argue later, not all backups are needed.

Algorithm 1 PBVI

1: $B \leftarrow \{b_0\}$
2: **while** true **do**
3: $\quad Improve(V,B)$
4: $\quad B \leftarrow Expand(B)$

Algorithm 2 Improve(V,B)

Input: V — a value function
Input: B — a set of belief points
1: **repeat**
2: $\quad$ **for each** $b \in B$ **do**
3: $\quad\quad \alpha \leftarrow backup(b)$
4: $\quad\quad add(V,\alpha)$
5: **until** V has converged

Spaan and Vlassis (2005) suggest to explore the world using a random walk from the initial belief state b_0. The points that were observed during the random walk compose the set B of belief points. The Perseus algorithm[2] (Algorithm 4) then iterates over these points in a random order. During each iteration backups are executed over points whose value has not yet improved in the current iteration.

[1]We also experimented with L_1 and $L_{\inf}$ and did not notice any improvement over L_2.

[2]We present here a single value function version of Perseus.

Algorithm 3 Expand(B)

Input: B — a set of belief points
1: $B' \leftarrow B$
2: **for each** $b \in B$ **do**
3: $\quad Successors(b) \leftarrow \{b' | \exists a, \exists o\ b' = \tau(b, a, o)\}$
4: $\quad B' \leftarrow B' \cup argmax_{b' \in Successors(b)} dist(B, b')$
5: **return** B'

The belief points used by Perseus are very different from the ones used by PBVI and in many cases most of them are redundant. The random walk Perseus uses is however much faster than the belief expansion of PBVI. The value function update may only execute backups over a small subset of the beliefs in B and yet ensures that the value for all points in B improves after each iteration. However, the behavior of Perseus is very stochastic. The random selections cause high variation in performance and in more complicated problems may cause the algorithm to converge very slowly. Nevertheless, the ideas pointed out by Spaan and Vlassis — eliminating the need for complete backups, and computing B rapidly — are an important foundation to our work.

Algorithm 4 Perseus

Input: B — a set of belief points
1: **repeat**
2: $\quad \tilde{B} \leftarrow B$
3: $\quad$ **while** $\tilde{B} \neq \phi$ **do**
4: $\quad\quad$ Choose $b \in \tilde{B}$
5: $\quad\quad \alpha \leftarrow backup(b)$
6: $\quad\quad$ **if** $\alpha \cdot b \geq V(b)$ **then**
7: $\quad\quad\quad \tilde{B} \leftarrow \{b \in \tilde{B} : \alpha \cdot b < V(b)\}$
8: $\quad\quad\quad add(V, \alpha)$
9: **until** V has converged

Smith and Simmons (2004; 2005) present the Heuristic Search Value Iteration (HSVI - Algorithm 5) that maintains both an upper bound and lower bound over the value function. HSVI traverses the belief space following the upper bound heuristic, greedily selecting successor belief points where the gap between the bounds is the largest, until some stopping criteria has been reached. Afterwards HSVI executes backups and H operator updates over the observed belief points on the explored path in a reversed order.

HSVI is stopped when the gap between bounds over the initial belief state b_0 is reduced to less than ϵ thus providing a guarantee over the quality of the value function. Even though Simth and Simmons prove that the gap is closed in a polynomial number of iterations, in most cases, closing this gap is impractical, especially due to the slow improvement of the upper bound. In practice HSVI computes good policies when the gap is still quite large.

Executing backups in a reversed order is important because the Bellman update uses the values of the successors to update the value of the current belief. As such, the value of a successor must be improved before the value of the current belief can be improved. Indeed, when backups in HSVI are done in order of detection the performance of HSVI is reduced drastically.

HSVI differs considerably from other point-based algorithms. First it collects new belief points after each iteration, as opposed to Perseus that uses a fixed set of points and PBVI that collects more points only if the current set was insufficient to produce a good policy. Second, the points that HSVI collects depend on the computed value function. As such, while it is possible to combine ideas from Perseus and PBVI, such as collect B following PBVI expansion and update the value function using the Perseus method, such combinations with HSVI are non trivial.

While producing very good trajectories in belief space, the computation of these trajectories is time consuming as it requires the complete expansion of all the successors of every belief state that is visited. Maintaining and updating the upper bound is also time consuming and provides an additional burden on HSVI.

Algorithm 5 HSVI

1: Initialize $\underline{V}$ and $\bar{V}$
2: **while** $\bar{V}(b_0) - \underline{V}(b_0) > \epsilon$ **do**
3: $\quad Explore(b_0, \underline{V}, \bar{V})$

Algorithm 6 Explore($b, \underline{V}, \bar{V}$)

Input: a belief state b, upper and lower bounds on the value function $\underline{V}, \bar{V}$.
1: **if** $\bar{V}(b) - \underline{V}(b) > \epsilon\gamma^{-t}$ **then**
2: $\quad a^* \leftarrow \text{argmax}_a\, Q_{\bar{V}}(b, a')$ (see Equation 8)
3: $\quad o^* \leftarrow \text{argmax}_o(\bar{V}(\tau(b, a, o)) - \underline{V}(\tau(b, a, o))$
4: $\quad Explore(\tau(b, a^*, o^*), \underline{V}, \bar{V})$
5: $\quad add(\underline{V}, backup(b, \underline{V}))$
6: $\quad \bar{V} \leftarrow HV(b)$

Recently, Shani et al. (Shani et al., 2007) suggested the Forward Search Value Iteration (FSVI) algorithm. FSVI uses ideas from HSVI, such as traversing the belief space following a heuristic and executing backups in a reversed order. The FSVI heuristic for traversing the belief space relies on an optimal Q function for the underlying MDP. This is a reasonable assumption as solving the underlying MDP is always easier than solving the POMDP. The algorithm simulates a traversal in both the MDP state space and the POMDP belief space, following always the best action for the MDP. As such, the traversal is ensured minimize the expected number of steps to the goal.

FSVI traversals are very fast to compute, requiring only $|A| + |S| + |O|$ operations for the heuristic computation at each step, compared to the $O(|S|^2)$ operations required just for the belief update. The downside of following an MDP-based heuristic is the inability to create traversals that visit states that may provide important observations, unless these states lie on some path from a start state to a goal, following the MDP policy.

As FSVI trajectories do not depend on the value function it computes, it is possible to break the process into first collecting

Algorithm 7 FSVI

```
1:  while Policy quality is insufficient do
2:     B ← {b_0}
3:     b ← b_0
4:     Choose s from the b_0 distribution
5:     while s is not a goal state do
6:        a* ← argmax_a Q(s, a)
7:        Choose s' from the tr(s, a*, ·) distribution
8:        Choose o from the O(a, s', ·) distribution
9:        b' ← τ(b, a*, o)
10:       Add b' to B
11:       b ← b'
12:       s ← s'
13:    Execute backups on B in reversed order
```

a set of belief points B, following a number of trajectories and maintaining the successor-predecessor relationship between belief states and after that computing a value function going over the trajectories in B in reverse order.

While in practice such an implementation is not useful, requiring additional memory for remembering the belief points, this view of FSVI allows us to better compare FSVI to Perseus, PBVI and the new algorithms suggested in this paper.

D. Other Related Work

Aside from point-based approaches, a second dominant method is the computation of a policy directly without a value function through the use of finite state controllers (see e.g. Poupart and Boutilier (2003)). A different approach for scaling up is the use of compression techniques to create a smaller model and solve the compressed POMDP instead of the larger one (Poupart & Boutilier, 2002). Yet another promising alternative is the use of bounded online search in belief space with heuristic functions to decide which action to execute in real-time (Paquet et al., 2005; Ross & Chaib-draa, 2007). Although interesting and useful, none of the above approaches have direct bearing to our approach, and thus we will not further discuss them here.

III. Enhancing Point Based Value Iteration

Point based algorithms compute a value function using α vectors by iterating over some finite set of belief points and executing a sequence of backup operations over these belief points. Our goal is to find the best possible policy within the minimal number of computations. A few factors may affect the amount of computations that is needed in order to compute a value function over a given domain:

- The number of belief points that are used for the computation of the value function. The number of belief points bounds the number of α-vectors in the value function which in turn influences the runtime of the backup process. The number of points is also important when considering the selection of the next point to improve.
- The number of backup operations. As backup operations are expensive, reducing the number of backup operations will reduce the execution time.
- Operations used to compute the belief states or to choose the next belief point to update. As explained above, the PBVI method for expanding the belief space and the HSVI method for selecting the next point in the traversal require a large number of operations, such as belief updates.

For both belief point selection and the ordering of backups, there is a need to balance the amount of operations required for computation and the gain from the reduction in belief set size and the number of backups.

In this paper we suggest methods that provide both a good selection of belief states and a good ordering of backups over the gathered belief states. We explain how to implement these methods so that the overall gain, balancing the additional required computations and the gain proves to be beneficial.

IV. Prioritized Point Based Value Iteration

We first discuss the proper ordering of backups over a given set of belief points B, attempting to find good policies as quickly as possible through point-based backups. We argue that many of the backups executed by point based methods are redundant. It is possible to achieve a policy with the same quality with a smaller amount of operations.

Given a predefined set of belief points, selecting such a sequence (or, more generally, a plan) of backup operations is a meta-reasoning problem. An optimal solution to this problem should greatly improve the speed of convergence. As the problem of selecting an optimal backup plan is a very hard problem, we use heuristics in order to attempt to find a good, but not necessarily optimal sequence. We will show empirically that even using a heuristic that results in backup sequences that are far from optimal, still significant runtime improvement is achieved over arbitrary orders.

A backup sequence seq_1 is better than sequence seq_2 if seq_1 is shorter than seq_2, and produces a policy which is no worse (we compare policies by measuring their ADR) than the one produced by seq_2. However, a sequence may be better but still induce an overhead that is intolerable. For example, we could have tried all the possible backup sequences, and afterwards selected the one that is shortest. Such an approach would give us the best possible sequence yet would require an absurdly large computation time (since this entails solving the POMDP problem of interest multiple times).

A better sequence, hence, does not always mean that the algorithm that uses it will be more efficient. It is possible that an algorithm executes a good sequence, but the time to compute it is much more that the execution of a worse sequence. Ideally we would like to obtain a good sequence rapidly.

We suggest creating better sequences using a heuristic that predicts useful backups. Clearly, the heuristic must be efficiently computable, so that the overhead of computing the heuristic does not outweigh any savings achieved by performing fewer backups.

Even if we ignore the needed effort for computing the sequence, the number of backups is still an inaccurate estimation of the actual execution time of the sequence. In practice,

backup execution time may differ due to their dependency on the size of the value function. It is likely that two different sequences of backups with identical lengths will produce value functions with different sizes and will therefore require different execution time. It is therefore better to evaluate the execution time in terms of g-operations or inner products. Nevertheless, for the clarity of the discussion, we measure high-level backups rather than the low level atomic g-operations.

A. Prioritizing MDP Solvers

A comparable scheme used for prioritizing in MDP solvers, suggests performing the next backup on the MDP state that maximizes the Bellman error:

$$e(s) = max_a[R(s,a) + \sum_{s'} tr(s,a,s')V(s')] - V(s). \quad (10)$$

$e(s)$ measures the change in the value of s from performing a backup. Wingate and Seppi (2005) present a very simple version of value iteration for MDPs using prioritization (Algorithm 8).

Algorithm 8 Prioritized Value Iteration for MDPs

1: $\forall s \in S, V(s) \leftarrow 0$
2: **while** V has not converged **do**
3: $\quad s \leftarrow \text{argmax}_{s' \in S}\, e(s')$
4: $\quad backup(s')$

A key observation for the efficiency of their algorithm is that after a backup operation for state s, the Bellman error recomputation need be performed only for the predecessors S' of s, defined as $S' = \{s' : \exists a, tr(s',a,s) \neq 0\}$. Hence, after initially setting $e(s) = \max_a R(s,a)$ for each $s \in S$, we update the priorities only for predecessors, avoiding a complete iteration through the state space.

B. Prioritizing POMDP Solvers

While the Bellman error generalizes well to POMDPs:

$$e(b) = max_a[r_a \cdot b + \sum_o pr(o|a,b)V(\tau(b,a,o))] - V(b) \quad (11)$$

there are two key differences between applying priorities to MDPs and POMDPs.

First, a backup update affects more than a single state. A new vector usually improves the local neighborhood of its witness belief point, but may improve the value for the entire belief space. As such, both the current value of any belief state may change following a backup operation, and the value of any of its successors. Hence, the error $e(b)$ for each belief state $b \in B$ may decrease or increase due to the new vector.

Second, the set of predecessors of a belief state cannot be efficiently computed, and its size is potentially unbounded. Consider for example a case where in some state s the agent receives a unique observation o, such that $O(*,s,o) = 1$ and $O(*,s',o) = 0$ for all $s' \neq s$. We denote by b_s the deterministic belief state of s s.t. $b_s(s) = 1.0$. Every belief state b such that $pr(b,*,o) > 0$, is therefore a predecessor of b_s. In the worst case the set of predecessors of a belief point is the entire belief simplex.

Moreover, even supposing that some similarity metric for finding the neighborhood of a belief point were defined, and that computation of the predecessor set were only for the finite set of belief points we use, directly applying the approach would still not be worthwhile. In practice, algorithms such as Perseus, frequently converge to an optimal solution while computing fewer backups than the number of belief points in the finite set. Pre-computations such as similarity matrices will take more time than the original algorithm they are designed to improve in the first place.

As we cannot find the set of belief states affected by the backup operation directly, we recompute the Bellman error for all belief states after every backup from scratch. When the number of belief points we use is relatively small this computation can be done without seriously damaging the performance. As the size of the problem — states, actions, observations and belief set size — increases, we can no longer afford the overhead of recomputing the Bellman error for all belief states.

We therefore take a stochastic approach, sampling a subset of the belief points set and computing the Bellman error only for this sampled subset. If the sampled subset does not contain a point with positive error, we sample again from the remaining subset (without repetitions) until a belief point with positive error is found. If there is no belief point with positive Bellman error then we assume that the value function has reached a fixed point and cannot be farther improved for the finite set of belief points.

While the upper bound complexity of both the backup operation and the error computation is identical, $O(|A||\Omega||S|^2)$, in practice, computing belief updates (Equation 1) is much faster than the computation of $g^{\alpha}_{a,o}$ (Equation 7). This is because belief states usually have many zero entries compared to α-vectors. Using data structures that support maintaining and iterating only over non-zero entries, all operations above can be implemented efficiently.

A new α-vector may change the Bellman error of many belief states in B, by changing both the value of any $b \in B$, and by changing the values of the successors of b. Nevertheless, when introducing a new α-vector, we do not need to run the entire computation $V(b) = \max_{\alpha \in V} \alpha \cdot b$. Given that for each cached belief state we also cache its latest optimal value, we can now check only whether the newest α-vector has improved the cached value.

When we update the Bellman error only over a sample of the belief states in B, the above approach needs to be generalized. We record for each α-vector in V the time it was inserted into V. Each belief state b is caching, aside for its current value $V(b)$, the timestamp $T(b)$ at which this value was last updated. When updating the value of a belief state b, only vectors that were inserted later than $T(b)$ are considered. We hence never compute the value $b \cdot \alpha$ for any b, α pair more than once.

As we use a finite, predefined set of belief states B, we can use caching in order to increase the computation efficiency at the cost of additional memory requirement. For each belief state in B we maintain a list of successors. As

a result, updating the Bellman error of a belief state b takes $O(|A||O||V_{new})$ where V_{new} is the number of α-vectors that were added after the last update of the Bellman error for b.

In the context of MDPs, the cost of maintaining a priority queue may induce an additional cost that annuls the benefits of the good order of backups (**?**). In our case we do not use a priority queue anyhow, since the priorities of all beliefs should be updated after each backup. Any suggested way to update the priorities of only a constant number of belief points, should also discuss the priority queue maintenance.

C. Prioritizing Existing Algorithms

We first show how prioritization can be used to enhance the performance of current algorithms. We suggest to replace the backup selection mechanism of existing algorithms with a prioritization scheme.

Prioritizing Perseus is straightforward. The "choose" step (Algorithm 4, line 4) is implemented in Perseus as a uniform selection among any of the current belief points inside $\tilde{B}$. Prioritized Perseus uses instead the Bellman error computation to choose a belief point whose value can be improved the most. As a result, backups are executed over belief states in order of reduced priorities. These priorities are updated after each backup, but belief points which were already improved in the current iteration are no longer considered.

PBVI improves its value function (Algorithm 1, line 3) by arbitrarily passing over all belief points and performing backup executions. We replace this inefficient computation of the 'Improve' operation with our PVI algorithm (see Section IV-D). As the number of points used by PBVI is relatively small, we did not use sampling when computing the Bellman error.

D. Prioritized Value Iteration

Finally, we present an independent algorithm — Prioritized Value Iteration (PVI). Like Perseus, PVI computes a value function over a fixed set of belief points collected before the algorithm is executed. However, Perseus operates in iterations over the set of belief points, attempting to improve all belief points between considering the same belief state twice. PVI considers at each step every possible belief state for improvement. It is likely, therefore, that some belief states will be backed up many times, while other belief states will never be used.

Algorithm 9 presents our PVI algorithm. The algorithm described here is the clean version of PVI, but in practice we implement the $argmax$ operation (line 2) using our sampling technique (Algorithm 10). If the algorithm is unable to find a belief state b with non-zero error (Choose returns nil), then we assume that the value function over B has converged.

If the prioritization metric is good, PVI executes a shorter sequence of backup operations. Indeed, experiments show that it uses significantly fewer backup operations than Perseus using our locally greedy Bellman error prioritization metric.

Algorithm 9 Prioritized Value Iteration

Input: B — a set of belief points
1: **while** V has not converged **do**
2: $b^* \leftarrow \text{argmax}_{b \in B} \, e(b)$
3: $\alpha \leftarrow backup(b^*)$
4: $add(V, \alpha)$

Algorithm 10 Choose

Input: B — a set of belief points, k — sample size
1: $B' \leftarrow B$
2: $b_{max} \leftarrow nil$
3: **while** B' not empty **do**
4: **for** $i = 0$ to k **do**
5: Select b with uniform distribution from B' and remove it
6: **if** $e(b) > e(b_{max})$ **then**
7: $b_{max} \leftarrow b$
8: **if** $e(b_{max}) > 0$ **then**
9: **return** b_{max}
10: **return** nil

V. Gathering Belief Points Through Heuristic Search

PBVI and Perseus use two opposing methods for gathering the belief point sets B; PBVI attempts to cover the reachable belief space in a uniform density by always selecting immediate successors that are as far as possible from the B. Perseus, on the other hand, simply explores the belief space by performing random trajectories. While the points gathered by PBVI generate a good B set (as shown later in our experiments), the time it takes to compute these points makes other algorithms more attractive.

The Perseus belief set is gathered very rapidly, and Perseus was shown to work well over small domains. In such small and medium sized domains it is possible to reach all interesting belief points through a random walk. In larger or more complex domains, however, it is unlikely that a random walk would visit every location where a reward can be obtained.

In the case where a sequence of actions is required to obtain a reward, while every deviation from the sequence causes the system to reset to its original state, it is unlikely that a random walk will be able to find the sequence. For example, if a robot is required to carry an object to some location, and place it there, dropping the object on the way might always force the robot to start over. It is also unlikely that the robot will know that it should put down the object it is carrying upon arriving at the destination.

We suggest to replace the random walk of Perseus and PVI with a heuristic search, based on the Q_{MDP} policy. The Q_{MDP} policy (Cassandra et al., 1994) uses the optimal Q-function Q^* of the underlying MDP to define a Q-function over the POMDP belief space:

$$Q_{MDP}(b, a) = \sum_s b(s) Q^*(s, a) \tag{12}$$

The POMDP policy is then defined by:

$$\pi_{Q_{MDP}}(b) = \text{argmax}_a \, Q_{MDP}(b, a) \tag{13}$$

A well known problem with MDP based heuristics for POMDP models is that MDP policies do not execute actions

that reduce the uncertainty of the belief. We overcome this difficulty by using an ϵ-greedy exploration heuristic — with probability $1-\epsilon$ choose the best action and with probability ϵ choose a randm action. The heuristic search allows us to use smaller, more focused sets of belief points, and thus, to reduce the runtime of our algorithms.

In our implementation we limit the heuristic selection to the set of actions, selecting observations from the O distribution. This setting is more appropriate for online algorithms where the agent actions are controlled, yet the observations are generated by the environment. However, in an offline setting it is possible to select the next observation through some heuristic too, as is done by HSVI. We leave discussion of good strategies for selecting observations for future research.

VI. Empirical Evaluations

A. Heuristic Search

We would first like to evaluate the benefits of our heuristic search method for gathering the belief set B over the random walk used by Perseus.

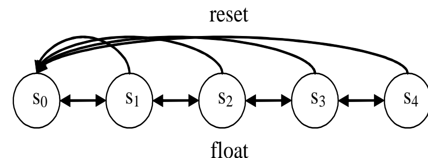
The float-reset problem can be used to model a scenario where a random walk has difficulties in achieving good performance. Consider n states connected in a chain. The system has 2 actions: float, which moves the agent with equal probability either up or down the chain, and reset, which sends the agent to the initial state. The agent receives a reward for executing reset at the last state in the chain. In the last state in the chain the agent receives a special observation. A good policy would be to float until the special observation is perceived and then activate the reset action. A random walk would take a very long time until such a specific sequence of actions is executed.

Figure 1 shows the number of steps a random walk requires to find the reward of the float-reset problem. A heuristic search found very rapidly the reward; with $n = 10$, all 10 runs of length 100 found the reward, with $n = 5$, all 10 runs of length 20 found the reward.

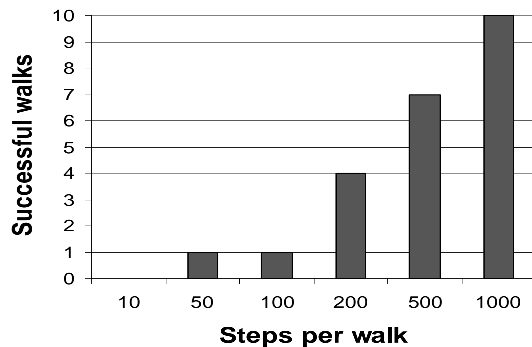
In other domains the agent may receive multiple rewards in different places. A random walk would need to visit all these places in order to allow Perseus to include these rewards in the POMDP value function. We experimented with two instances of the RockSample (Smith & Simmons, 2005) domain, both with an 8×8 board, one with 4 rocks and one with 6 rocks. Sampling every rock results in a different reward. The belief set gathering process must pass through the locations of the rocks on the board. Figure 2 shows how the size of the belief set influences the number of rocks visited during the gathering process and hence, the ADR.

B. Reducing Backup Sequences

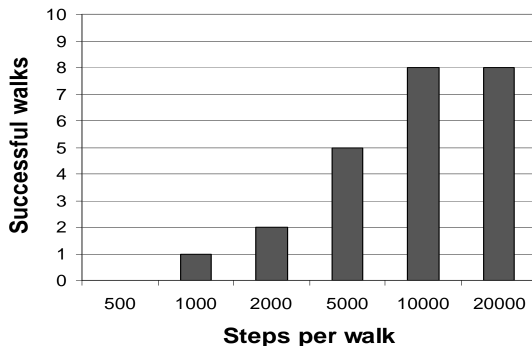
In addition to showing that our methods achieve significant runtime improvement (see the coming sections), we would like to begin addressing the meta-reasoning issue of optimal backup sequences. In this way, one could examine the potential gain that could be achieved by methods of prioritizing backup operations, regardless of whether it is achieved in practice. This would also give us another yardstick for evaluating the quality of our prioritization heuristics. Examining prioritization heuristics in this light may lead to ideas of how to improve them in the future.



(a) The Float reset problem with $n = 5$ states.



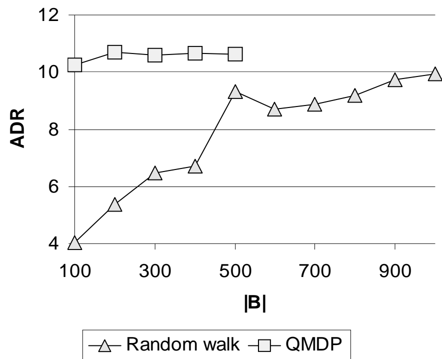
(b) Results for float reset with 5 states.



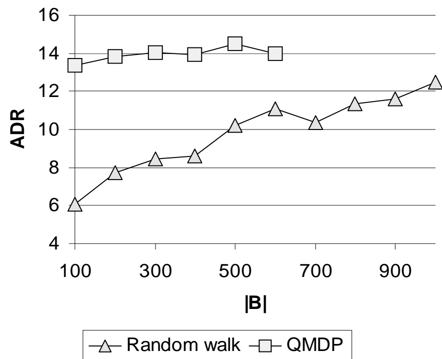
(c) Results for float reset with 10 states.

Fig. 1. Number of times, out of 10 runs, a random walk succeeds to find the goal within the specified number of steps over the float-rest problem with $n = 5$ (a) and $n = 10$ (b).

However, solving this meta-reasoning problem is very hard. Even knowing the solution to a POMDP, and even in retrospect, after observing algorithm executions, it is very hard to come up with an optimal, or even provably near-optimal backup sequences. Yet, the retrospective view allows us at least to look at sequences of backups for a POMDP instance (and for a specific choice of belief points), and assess which of the backup operations seem to have moved the algorithm to convergence faster. We could in principle attempt to re-run the algorithm for every possible sub-sequence, and examine the quality of each such sub-sequence w.r.t. length of the sub-sequence and resulting policy ADR, and treat subsequences that are optimal in this respect as approximately optimal. (Note that we could either weight sequence length in with ADR, or alternately generate an optimal performance profile, i.e. what is the best ADR reached for subsequences of each given length).



(a) Results for RockSample 8×8 and 4 rocks.



(b) Results for RockSample 8×8 and 6 rocks.

Fig. 2. ADR computed by Perseus as a function of the belief set size over the RockSample problem with an 8×8 board and 4 rocks (a) or 6 rocks (b). Belief sets are computed by a random walk or the Q_{MDP} heuristic.

Unfortunately, since trying all subsequences, even off-line, is prohibitively expensive (we also need to try all possible permutations!), in practice we use the following scheme: compute a backup sequence by a point-based algorithm over a fixed set of belief points. Each backup changes the value of the belief point it was executed upon. We can run the sequence again, filtering out backups that contributed a value change of no more than ϵ, but otherwise maintaining the original order of backups, and check the resulting ADR. The new sequence may, once again, contain backups that improved the value by less than ϵ. We hence repeat this process until no more backups can be removed, noting the number of remaining backups and the ADR after each iteration. Once all backups improve the value of the updated belief point by at least ϵ we double ϵ and continue the experiment until all backups have been removed.

We evaluate the possible reduction in the sequence of backups that three methods — PBVI, Perseus, and FSVI — execute over a finite set B. We therefore gathered a belief set B of size 100 using our heuristic search and executed for each method 1000 backups. PBVI backups were executed in the order the belief states were discovered (fixed, arbitrary order), Perseus backups were executed using the standard Perseus procedure. To simulate FSVI we chose from B a belief state where a heuristic search traversal terminated, and moved back towards its successors until the initial belief state was reached. While our heuristic traversal is not identical to the traversal FSVI uses, it still allows us to estimate the reversed order of backups.

We used 5 different belief point sets and for each of these sets we ran 5 value function computation trials of FSVI and Perseus. PBVI that executes a non random sequence of backups was executed only once. We then run the backup pruning procedure explained above and averaged over the different trials.

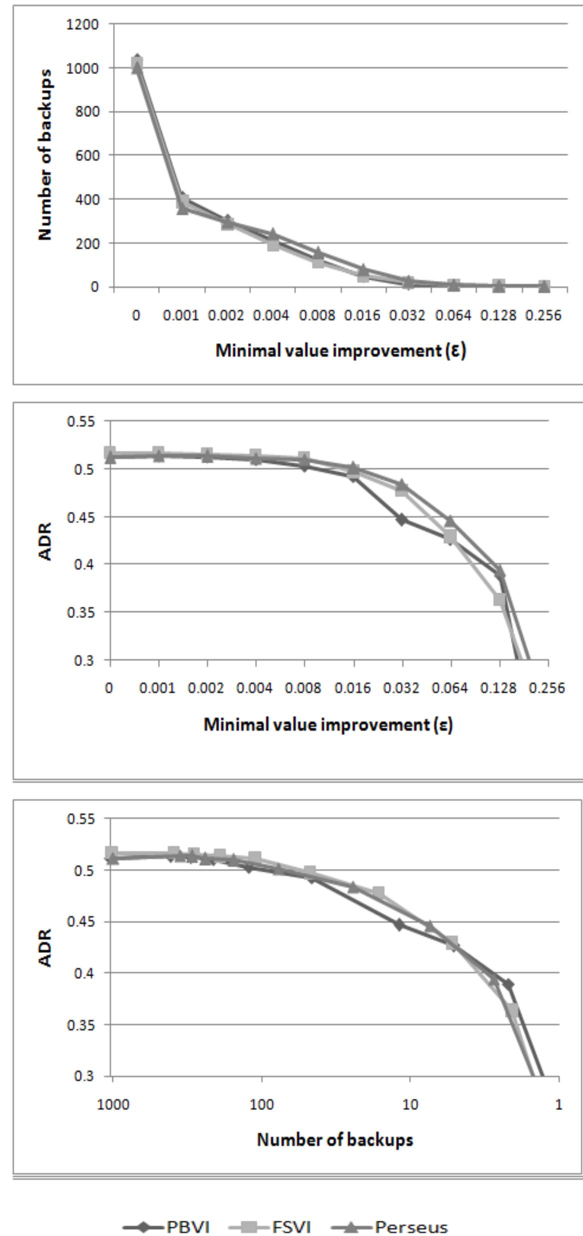


Fig. 3. Pruning backups with local value improvement of less than ϵ on the Hallway domain.

As Figure 3 clearly show for the Hallway problem, the local improvement of a belief state update is a good estimator as to its usability. When pruning backups that added no more than 0.001, there is no noticeable decrease in value function update in any of the algorithms we have tested. In fact, in some cases removing such backups even improves the value function

quality. As we start removing backups that contributed higher value improvements, the value function slowly begins to degrade. Our experiments provide farther evidence that a value function of equal quality can be computed using much fewer backups than the algorithms use in practice.

Another interesting aspect that is demonstrated in Figure 3 is that all the algorithms present similar sensitivity to the Bellman error. Even though the algorithms produce very different sequences of backups, in all cases removing backups with small Bellman error has little effect over the quality of the value function. This farther supports our claim that there is a high correlation between the Bellman error and the importance of a backup. While this might seem obvious at first, this is not so. A backup produces an α-vector that may define a new value for many belief points. It is theoretically possible that a backup may only slightly improve the value for the updated belief point, but will contribute much to many other belief points. However, as our results seem to indicate, in practice this is not the case.

Note that this type of empirical examination, which is very informative, cannot, obviously, be done during the run of the algorithm. However, it does seem that the direction we took, avoiding backups that have a low local impact is useful, and that farther effort should be made to discover ways to rapidly provide estimation as to the effect of a value update over a specific belief point. Although the above retrospective, "clairvoyant" backup-sequence optimization scheme is not a reasonable runtime meta-reasoning scheme, we can still check how our algorithms perform against this idealized yardstick.

We have done so only for the Hallway domain, due to the excessive runtime involved for each data point, with results shown in Figure 4. For this small problem, it seems that our PVI algorithm selected a backup sequence that was very near to the optimal, except when only allowed fewer than 25 backup operations. However, despite the fact that our algorithms did better than competing algorithms (see below), the backup sequences used were not so close to optimal for the more complicated problem instances.

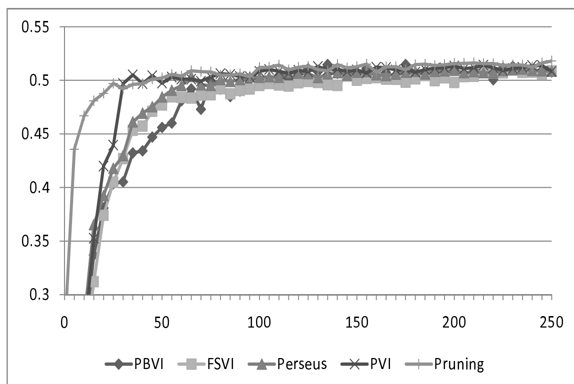


Fig. 4. Convergence of various algorithms, compared to the omnipotent clairvoyant backup pruning method

C. Improved Evaluation Metrics

Previous researchers (Brafman, 1997; Pineau et al., 2003; Smith & Simmons, 2005; Spaan & Vlassis, 2005; Paquet et al., 2005) limit their reported results to execution time, average discounted reward and in some cases the number of vectors in the resulting value function.

Value function evaluation — Average discounted reward (ADR) is computed by simulating the agent interaction with the environment over a number of steps (called a *trial*) and averaging over a number of different trials:

$$\frac{\sum_{i=0}^{\#trials} \sum_{j=0}^{\#steps} \gamma^j r_j}{\#trials} \tag{14}$$

ADR is widely agreed as a good evaluation of the quality of a value function.

ADR is however very noisy when the number of trials or the number of steps is too small. For example, on the Hallway example, with 250 trials per ADR, we observed a noise of about 0.5 (about 10% of the optimal performance) while with $10,000$ trial the noise dropped to around 0.0015.

In our experiments we interrupted the executed algorithms occasionally to check the ADR of the current value function. We have observed that in some cases, an algorithm managed to produce a value function providing surprisingly good ADR, but additional backups caused a degradation in ADR. We treat such cases as a noise in the convergence of the value function. To decrease this noise we used a filter.

The actual filter used in order to decide on the convergence threshold for results shown in the table was the first order filter with weight 0.5 —

$$FADR_i = 0.5 \times ADR + 0.5 \times FADR_{i-1} \tag{15}$$

where $FADR_0 = 0$. This filter does not provide any guarantee for the accuracy of the result, and as such it is not optimal. Nevertheless it is a (minor) improvement over related research that used only a single noisy ADR measurement. The algorithm was stopped once the filtered ADR has exceeded a predefined target.

Execution time — It was observed before that execution time is a poor estimate of the performance of the algorithm. Execution time is subject to many parameters irrelevant to the algorithm itself, such as the machine and platform used to execute it, the programming language, the level of implementation and so forth. It is also important to report CPU time rather than wall-clock time.

As all algorithms discussed in this paper compute a value function using identical operations such as backups, τ function computations, and inner products $(\alpha \cdot b)$, it seems that recording the number of executions of those basic building blocks of the algorithm is more informative.

Backup operations themselves do not make a good estimator because they depend on the number of vectors in the current value function. A better estimation is hence the $g_{a,o}^{\alpha}$ computation (Equation 7), which depends only on the system dynamics.

Memory — While the size of the computed value function is a good estimate for the execution time of the resulting policy. It can also be used to estimate the memory capacity required for the computation of the algorithm.

A second indication for the amount of required memory is the number of maintained belief points throughout the

execution of the algorithm. As some operations (e.g. the Bellman error computation) can be highly improved when caching more belief states, we also report the number of belief states an algorithm caches.

D. Experimental Setup

In order to test our prioritized approach, we tested all algorithms on the full set of standard benchmarks from the point-based literature: Hallway, Hallway2 (Littman et al., 1995), TagAvoid (Pineau et al., 2003) and RockSample (Smith & Simmons, 2004). Table I contains the problem measurements for the benchmarks including the size of the state space, action space and observation space, the number of belief points in the set $|B|$ used for Perseus, Prioritized Perseus and PVI, and the error in measuring the ADR over $10,000$ trials for each problem.

Problem	$\lvert S\rvert$	$\lvert A\rvert$	$\lvert O\rvert$	$\lvert B\rvert$	ADR Error
Hallway	61	5	21	250	±0.0015
Hallway2	93	5	17	300	±0.004
Tag Avoid	870	5	30	350	±0.045
Rock Sample 4,4	257	9	2	500	±0.075
Rock Sample 5,5	801	10	2	500	±0.3
Rock Sample 5,7	3201	12	2	500	±0.25

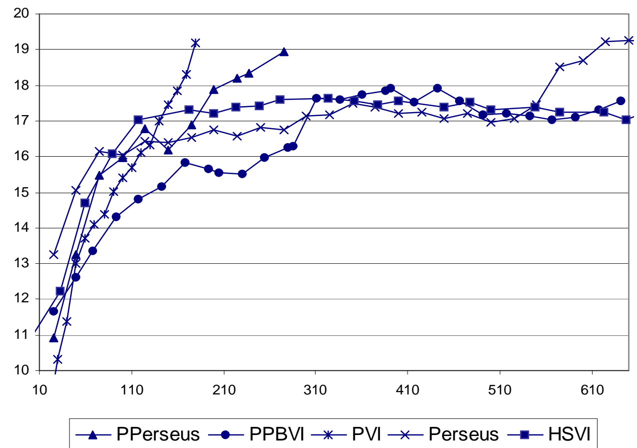
TABLE I
BENCHMARK PROBLEM PARAMETERS

We implemented in Java a standard framework that incorporated all the basic operators used by all algorithms such as vector inner products, backup operations, τ function and so forth. All reported results were gathered by executing the algorithms on identical machines —- $x86$ 64-bit machines, dual-proc, processor speed 2.6Ghz, 4Gb memory, 2Mb cache, running Linux and JRE 1.5.

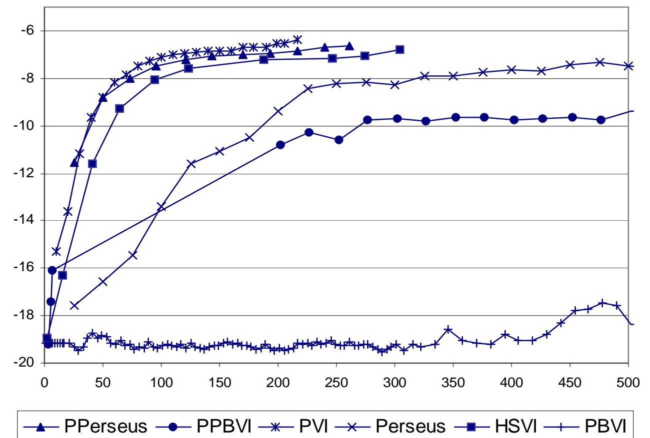
As previous researchers have already shown the maximal ADR achievable by their methods, we focus our attention on convergence speed of the value function to the reported ADR. We executed all algorithms, interrupting them from time to time in order to compute the efficiency of the current value function using ADR over 5000 trials. Once the filtered ADR has reached the maximal value reported in past publications execution was stopped. The reported ADR was then measured over additional $10,000$ trials (error in measurement is reported in Table I).

For algorithms that require a given set of belief states B — Perseus, Prioritized Perseus and PVI — we pre-computed 5 different sets of belief points for each problem. Each belief points set was computed by simulating an interaction with the system following the Q_{MDP} policy with an ϵ-greedy exploration factor ($\epsilon = 0.1$). For each such belief points set we ran 5 different executions with different random seeds resulting in 25 different runs for each stochastic method. The number of belief points used for each problem is specified in Table I. Using the Q_{MDP} heuristic for gathering belief points allowed us to use a considerably smaller belief set than the original Perseus algorithm (Spaan & Vlassis, 2005).

Algorithms that are deterministic by nature — PBVI, Prioritized PBVI and HSVI — were executed once per problem.



(a) Rock Sample 5,5



(b) Tag Avoid

Fig. 5. Convergence of various algorithms on the Rock Sample 5,5 problem (a) and the Tag Avoid problem (b). The X axis shows the number of backups while the Y axis shows the ADR.

E. Results

Table II presents our experimental results. For each problem and method we report:

1) Resulting ADR
2) Size of the final value function ($|V|$)
3) CPU time until convergence
4) The number of backups
5) The number of $g^{\alpha}_{a,o}$ operations
6) The number of computed belief states
7) the number of τ function computations
8) The number of inner product operations.

The reported numbers do not include the repeated expensive computation of the ADR, or the initialization time (identical for all algorithms). Results for algorithms that require a pre-computed belief space do not include the effort needed for this pre-computation. We note, however, that it took only a few seconds (less than 3) to compute the belief subset B over all problems.

To better illustrate the convergence of the algorithms we have also plotted the convergence of the ADR vs. the number

Method	ADR	$\|V\|$	Time (secs)	#Backups	$\#g^{\alpha}_{a,o} \times 10^6$	#belief states$\times 10^4$	$\#\tau \times 10^3$	$\#\alpha \cdot b \times 10^6$
Hallway								
PVI	0.517±0.0027	144±32	**75**±32	**504**±107	3.87±1.75	1.99±0.04	4.8±9.8	13.11±5.19
PPerseus	0.517±0.0025	173±43	126±47	607±166	5.52±2.95	1.99±0.04	4.8±9.8	26.87±9.2
Perseus	0.517±0.0024	466±164	125±110	1456±388	31.56±27.03	0.03±0	0±0	32.07±27.16
PPBVI	0.519	235	95	725	9.09	1.49	13.45	25.72
PBVI	0.517	253	118	3959	31.49	1.49	15.79	31.69
HSVI	0.516	182	314	634	5.85	3.4	34.52	6.67
Hallway2								
PVI	0.344±0.0037	234±32	75±20	262±43	2.59±0.84	2.49±0.11	5.47±9.99	6.96±2.01
Pperseus	0.346±0.0036	273±116	219±155	343±173	4.76±5.48	2.49±0.11	5.25±9.99	18.97±12.79
Perseus	0.344±0.0034	578±95	134±48	703±120	17.03±6.08	0.03±0	0±0	17.31±6.13
PPBVI	0.347	109	**59**	**137**	0.61	2.03	10.77	4.22
PBVI	0.345	128	76	1279	7.96	1.52	5.59	8.02
HSVI	0.341	172	99	217	1.56	2.11	11.07	1.81
Tag Avoid								
PVI	-6.467±0.19	204±38	**40**±12	211±38	0.42±0.19	0.16±0	0.5±1.02	0.95±0.25
PPerseus	-6.387±0.18	260±43	105±26	265±44	5.27±1.8	1.73±0.02	5.68±11.59	12.82±3.01
Perseus	-6.525±0.20	365±69	212±174	11242±10433	28.69±32.09	0.04±0	0±0	30.78±33.96
PPBVI	-6.271	167	50	**168**	2.09	0.41	32.11	2.45
PBVI	-6.6	179	1075	21708	407.04	0.41	56.53	409.5
HSVI	-6.418	100	52	304	0.5	0.29	1.74	0.53
Rock Sample 4,4								
PVI	17.725±0.32	231±41	**4**±2	232±42	0.36±0.14	0.41±0.01	1.17±2.38	1.74±0.42
PPerseus	17.574±0.35	229±26	5±2	228±27	0.34±0.08	0.41±0.01	1.17±2.38	1.91±0.29
Perseus	16.843±0.18	193±24	158±33	24772±5133	59.96±13.06	0.05±0	0±0	66.52±14.25
PPBVI	18.036	256	229	265	0.62	2.43	55.31	9.46
PBVI	18.036	179	442	52190	113.16	1.24	35.47	119.8
HSVI	18.036	123	**4**	**207**	1.08	0.1	1.17	1.09
Rock Sample 5,5								
PVI	19.238±0.07	357±56	21±7	362±63	0.99±0.36	0.46±0.01	1.37±2.79	3.39±0.87
PPerseus	19.151±0.33	340±53	**20**±6	**339**±53	0.88±0.28	0.46±0.01	1.37±2.79	3.5±0.73
Perseus	19.08±0.36	413±56	228±252	10333±9777	60.34±66.62	0.05±0	0±0	63.17±69.21
PPBVI*	17.97	694	233	710	4.95	0.95	17.97	18.12
PBVI*	17.985	353	427	20616	72.01	0.49	11.28	75.64
HSVI	18.74	348	85	2309	10.39	0.26	2.34	10.5
Rock Sample 5,7								
PVI	22.945±0.41	358±88	**89**±34	359±89	1.28±0.64	0.29±0.01	0.73±1.49	2.98±1.16
Pperseus	22.937±0.70	408±77	118±37	407±77	1.61±0.59	0.29±0.01	0.73±1.49	4.09±0.98
Perseus	23.014±0.77	462±70	116±31	1002±195	5.18±1.9	0.02±0	0±0	5.36±1.93
PPBVI*	21.758	255	117	**254**	0.61	0.23	2.71	1.59
PBVI*	22.038	99	167	2620	3.05	0.15	1.66	3.23
HSVI	23.245	207	156	314	0.83	0.71	4.2	0.88

TABLE II

PERFORMANCE MEASUREMENTS. THE ALGORITHMS THAT EXECUTED FEWER BACKUPS AND CONVERGED FASTER ARE BOLDED.

of backups an algorithm performs in Figure 5[3]. The graphs contain data collected over separate executions with fewer trials (500 instead of 10000) so Table II has more accurate results.

HSVI is the only method that also maintains an upper bound over the value function ($\bar{V}$). Table III contains additional measurements for the computation of the upper bound: the number of points in $\bar{V}$, the number of projections of other points onto the upper bound, and the number of upper bound updates ($HV(b)$ — Equation 9).

For the stochastic methods we show standard deviations over the 25 runs for all categories.

PBVI and PPBVI failed in two cases (Rock Sample 5,5 and Rock Sample 5,7) to improve the reported ADR even when allowed more time to converge. These rows are marked with an asterix.

As our PVI must update the Bellman errors for the belief states after each new α vector is computed, it is highly affected by the size of the belief set B. Table IV shows the effect of the size of the B over the time it takes for the algorithm to run and over the ADR in the Hallway and Hallway2 domains. We ran PVI for 200 backups, stopping after each 50 backups to evaluate the execution time and ADR[4]. We report results both without sampling and with sampling of 25 belief points

Problem	$\|\bar{V}\|$	Upper bound projections	#HV(b)	$\|B\|$
Hallway	423	106132	1268	523
Hallway2	232	37200	434	171
Tag Avoid	1101	29316	1635	248
Rock Sample 4,4	344	6065	414	176
Rock Sample 5,5	801	101093	6385	1883
Rock Sample 5,7	3426	9532	628	268

TABLE III

UPPER BOUND MEASUREMENTS FOR HSVI

[3] In Figure 5a, PBVI was removed as its ADR was below the minimal value displayed in the graph.

[4] Time was measured on a different machine than the results in Table II. ADR was computed over 1000 trials.

	$\lvert B \rvert = 25$		$\lvert B \rvert = 50$		$\lvert B \rvert = 100$		$\lvert B \rvert = 250$		$\lvert B \rvert = 500$	
Backups	**ADR**	**Time**	**ADR**	**Time**	**ADR**	**Time**	**ADR**	**Time**	**ADR**	**Time**
Hallway - no sampling										
50	0.491	3.6	0.498	6.8	0.497	18.8	0.498	78.6	0.498	78.6
100	0.482	7.6	0.502	13	0.509	35	0.51	144	0.51	144
150	0.49	11.8	0.508	20.4	0.504	53	0.517	212.4	0.517	212.4
200	0.491	16	0.511	27.4	0.509	74.8	0.514	283.6	0.514	283.6
Hallway - sampling 25 points										
50	×	×	0.49	4.6	0.482	6.2	0.497	9.6	0.505	15.8
100	×	×	0.505	8.6	0.505	11.6	0.508	18.2	0.505	29
150	×	×	0.508	13.6	0.502	18.2	0.516	27.2	0.514	42
200	×	×	0.509	18.2	0.515	24.4	0.51	36.4	0.515	55.4
Hallway2 - no sampling										
50	0.286	5.6	0.286	10.8	0.311	23.8	0.314	63	0.308	127.6
100	0.3	12.6	0.299	23.6	0.316	49.8	0.323	140.2	0.318	274.2
150	0.309	21.4	0.307	38.2	0.326	77.4	0.326	224.6	0.331	447.4
200	0.309	31	0.298	54.8	0.332	108.4	0.33	320.6	0.32	638
Hallway2 - sampling 25 points										
50	×	×	0.294	7.4	0.308	10.4	0.304	18.2	0.314	27.4
100	×	×	0.3	15.4	0.325	21	0.319	37.2	0.319	56.4
150	×	×	0.311	24.6	0.324	33	0.321	58.4	0.323	88.6
250	×	×	0.298	35.2	0.323	45.8	0.327	80	0.321	121.2

TABLE IV
INFLUENCE OF THE BELIEF SET SIZE OVER THE EXECUTION TIME (SECONDS) OF PVI

at each step.

F. Discussion

Our results clearly show how selecting the order by which backups are performed over a predefined set of points improves the convergence speed. When comparing PBVI to Prioritized PBVI and Perseus to Prioritized Perseus, we see that our heuristic selection of backups leads to considerable improvement in runtime. This is farther demonstrated by the new PVI algorithm. In all these cases, there is an order of magnitude reduction in the number of backup operations when the next backup to perform is chosen in an informed manner. However, we also see that there is a penalty we pay for computing the Bellman error, so that the saving in backups does not fully manifest in execution time. Nevertheless, this investment is well worth it, as the overall performance improvement is clear. Although the ADR to which the different algorithms converge is not identical, the differences are minor, never exceeding 2%, making their ultimate ADR equivalent, for all practical purposes.

Examining Table II, we see that our PVI algorithm results in convergence time that is at least comparable, if not better, than existing point based methods (PBVI, Perseus and HSVI). The efficiency of its backup choices shows up nicely in Figure 5, where we see the steep improvement curve of PVI.

In many case, HSVI also executes a smaller number of backups than other algorithms. Indeed, one may consider HSVI's selection of belief space trajectories as a method for backup sequence computation and hence, as a prioritization metric. Nevertheless, in most cases our form of backup selection exhibits superior runtime to HSVI, even when the number of backups HSVI uses is smaller. This is due to the costly maintenance of the upper bound over the value function.

The good performance of HSVI is due to two factors — the heuristic selection of belief space trajectories, and the ordering of backups over the belief states in the trajectory. To test this claim, we have modified HSVI, executing backups *before* computing the next belief point in the trajectory. As expected, this caused HSVI to slow down considerably and made it unsuitable to solving even medium sized problems. HSVI also suffers from the same problem as our PVI — the computation time of the order of backups is time consuming. Indeed, FSVI (Shani et al., 2007) is a trial based algorithm similar to HSVI. FSVI uses more backups due to a less focused heuristic search, yet requires almost no time to compute this heuristic. As a result, in most cases FSVI computes policies much faster than HSVI. As FSVI uses the underlying MDP to guide its exploration, it is not guaranteed to find a good solution. Specifically, in tasks that require multiple actions to reduce the partial observability, FSVI cannot compute good policies.

A new approach within the PVI framework suggested in this paper was recently introduced as the SCVI algorithm (Virin et al., 2007). SCVI uses a less focused heuristic than the Bellman error, but allows the rapid computation of the order of backups. This algorithm also uses a fixed set of belief points B. SCVI clusters B using optimal MDP state values and iterates over clusters by decreasing values. As such, SCVI captures the successor-predecessor relationship of belief points without explicitly computing it for each pair of beliefs.

SCVI executes in most cases more backups than PVI, but computes good value functions much faster. This farther supports the main claim of this paper, that the order of backups is crucial to the speed of convergence. Other such heuristics probably exist that lie, as SCVI does, in the range between the arbitrary backup sequence of PBVI which requires no computations but is extremely inefficient, and the Bellman error proposed by PVI that provides short backup sequences but demands extensive computations.

Our heuristic belief states gathering process is suitable for algorithms, such as Perseus and PVI, that first compute a belief subset B and then compute a value function over B. PBVI,

HSVI and FSVI take a different approach. These algorithms interleave belief selection and value function updates. Such an iterative process can allow the algorithm to select points based on the current value function, an approach that is indeed implemented by HSVI. Interleaving belief space sampling and value function updates can lead to superior results to the fixed belief set we use in this paper, but we leave such discussion to future research.

VII. Conclusions

This paper demonstrates how point-based POMDP solvers such as PBVI and Perseus can greatly benefit from intelligent selection of the order of backup operations, and that such selection can be performed efficiently, so that the overall performance of the algorithms improves. It also presents an independent algorithm — Prioritized Value Iteration (PVI) — that outperforms all previous point-based algorithms on a large set of benchmarks converging faster toward comparable values of ADR. The extensive experimental results reported here provide a clearer picture of different aspects of the performance of PVI and current point-based algorithms on popular domains from the literature.

All the prioritized algorithms described in this paper use the same heuristic measure, the Bellman error, to decide on the sequence of backups. The method for selecting the order of backups using the Bellman error is pointwise greedy. While this choice may be optimal in the context of MDPs, in the context of POMDPs it does not take into account the possible improvement of a backup over other belief points as well. It is quite likely that executing a backup that improves a region of the belief space rather than a single belief point may have better influence over the convergence of the value function. Thus, future work should examine other possible heuristic functions that take this into account. The Bellman error is also expensive to compute, forcing us to estimate only a sampled subset of the belief points. This implies that cheaper alternatives that lead to similar quality of backup selection may lead to algorithms that are an order of magnitude faster than current algorithms.

We have also presented a new method for belief state selection for algorithms such as Perseus and PVI that first select a set of belief points and then compute a value function only for this fixed set. We show our Q_{MDP} based heuristic to provide belief points that are smaller and more focused and thus, to increase the execution time of the algorithms.

References

Bonet, B., & Gefner, H. (2003). Labeled RTDP: Improving the convergence of real-time dynamic programming. *ICAPS* (pp. 12–31).

Brafman, R. I. (1997). A heuristic variable grid solution method for pomdps. *AAAI'97*.

Cassandra, A. R., Kaelbling, L. P., & Littman, M. L. (1994). Acting optimally in partially observable stochastic domains. *AAAI'94* (pp. 1023–1028).

Cassandra, A. R., Littman, M. L., & Zhang, N. (1997). Incremental pruning: A simple, fast, exact method for partially observable markov decision processes. *UAI'97* (pp. 54–61).

Littman, M. L., Cassandra, A. R., & Kaelbling, L. P. (1995). Learning policies for partially observable environments: Scaling up. *ICML'95*.

Lovejoy, W. S. (1991). Computationally feasible bounds for partially observable markov decison processes. *Operations Research*, *39*, 175–192.

Moore, A. W., & Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Journal of Machine Learning*, *13*, 103–130.

Paquet, S., Tobin, L., & Chaib-draa, B. (2005). Real-time decision making for large pomdps. *AI'2005*.

Pineau, J., Gordon, G., & Thrun, S. (2003). Point-based value iteration: An anytime algorithm for POMDPs. *IJCAI*.

Poupart, P., & Boutilier, C. (2002). Value-directed compression of POMDPs. *NIPS 15*. MIT Press.

Poupart, P., & Boutilier, C. (2003). Bounded finite state controllers. *NIPS 16*.

Poupart, P., & Boutilier, C. (2004). VDCBPI: an approximate scalable algorithm for large POMDPs. *NIPS 17*.

Ross, S., & Chaib-draa, B. (2007). Aems : An anytime online search algorithm for approximate policy refinement in large pomdps. *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*.

Shani, G., Brafman, R., & Shimony, S. (2006). Prioritizing point-based pomdp solvers. *ECML*.

Shani, G., Brafman, R., & Shimony, S. (2007). Forward search value iteration for pomdps. *IJCAI-07*.

Smallwood, R., & Sondik, E. (1973). The optimal control of partially observable processes over a finite horizon. *OR*, *21*.

Smith, T., & Simmons, R. (2004). Heuristic search value iteration for pomdps. *UAI 2004*.

Smith, T., & Simmons, R. (2005). Point-based pomdp algorithms: Improved analysis and implementation. *UAI 2005*.

Spaan, M. T. J., & Vlassis, N. (2005). Perseus: Randomized point-based value iteration for POMDPs. *JAIR*, *24*, 195–220.

Virin, Y., Shani, G., Shimony, S., & Brafman, R. (2007). Scaling up: Solving pomdps through value based clustering. *AAAI*.

Wingate, D., & Seppi, K. D. (2005). Prioritization methods for accelerating mdp solvers. *JMLR*, *6*, 851–881.