UNIVERSITEIT
VAN
AMSTERDAM

IAS technical report IAS-UVA-04-02

Perseus: randomized point-based value iteration for POMDPs

Matthijs T. J. Spaan and Nikos Vlassis
Informatics Institute
Faculty of Science
University of Amsterdam
The Netherlands

Partially observable Markov decision processes (POMDPs) form an attractive and principled framework for agent planning under uncertainty. Point-based approximate techniques for POMDPs compute a policy based on a finite set of points collected in advance from the agent's belief space. We present a randomized point-based value iteration algorithm called PERSEUS. The algorithm performs approximate value backup stages, ensuring that in each backup stage the value of all points in the belief set is improved (or at least does not decrease). Contrary to other point-based methods, PERSEUS backs up only a (random) subset of belief points—the key observation is that a single backup may improve the value of many points in the set. We show how the same idea can be extended to dealing with continuous action spaces. Experimental results show the potential of PERSEUS in large scale POMDP problems.

IAS
intelligent autonomous systems

Contents

Intelligent Autonomous Systems
Informatics Institute, Faculty of Science
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam
The Netherlands
Tel (fax): +31 20 525 7461 (7490)
http://www.science.uva.nl/research/ias/

Corresponding author:
Matthijs T. J. Spaan
tel: +31 20 525 7524
mtjspaan@science.uva.nl
http://www.science.uva.nl/~mtjspaan/

1 Introduction

A major goal of Artificial Intelligence (AI) is to build intelligent agents [29]. The agent, whether physical or simulated, should be able to autonomously perform a given task. An intelligent agent is often characterized by its sense–think–act loop: it uses sensors to observe the environment, considers this information to decide what to do, and executes the chosen action. The agent influences its environment by acting and can detect the effect of its actions by sensing: the environment closes the loop. In this work we are interested in computing a *plan* that maps sensory input to the optimal action to execute for a given task. We consider types of domains in which an agent is uncertain about the exact effect of its actions. Furthermore, it cannot determine with full certainty the state of the environment with a single sensor reading, i.e., the environment is only partially observable to the agent.

Planning under these kinds of uncertainty is a challenging problem as it requires reasoning over all possible futures given all possible histories. Partially observable Markov decision processes (POMDPs) provide a rich mathematical framework for acting optimally in such partially observable and stochastic environments [3, 32, 15, 13]. The POMDP defines a sensor model specifying the probability of observing a particular sensor reading in a specific state and a stochastic transition model which captures the uncertain outcome of executing an action. The agent's task is defined by the reward it receives at each time step and its goal is to maximize the discounted cumulative reward. Assuming discrete models, the POMDP framework allows for capturing all uncertainty introduced by the transition and observation model by defining and operating on the *belief state* of an agent. A belief state is a probability distribution over all states and summarizes all information regarding the past.

Using belief states allows one to transform the original discrete state POMDP into a continuous state Markov decision process (MDP), which can in turn be solved by corresponding MDP techniques [6]. However, the optimal value function in a POMDP exhibits particular structure (it is piecewise linear and convex) that one can exploit in order to facilitate the solving. Value iteration, for instance, is a method for solving POMDPs that builds a sequence of value function estimates which converge to the optimal value function for the current task [32]. The value function is parametrized by a finite number of hyperplanes, or vectors, over the belief space, and which partition the belief space in a finite amount of regions. Each vector maximizes the value function in a certain region, and with each vector an action is associated which is the optimal action to take for beliefs in its region. Computing the next value function estimate—looking one step deeper into the future—requires taking into account all possible actions the agent can take and all subsequent observations it may receive. Unfortunately, this leads to an exponential growth of vectors with the planning horizon. Many of the computed vectors will be useless in the sense that their maximizing region is empty, but identifying and subsequently pruning them is an expensive operation.

Exact value iteration algorithms [32, 11, 13] search in each value iteration step the complete belief simplex for a minimal set of belief points that generate the necessary set of vectors for the next horizon value function. This typically requires solving a number of linear programs and is therefore costly in high dimensions. In [38] it was argued that value iteration still converges to the optimal value function if exact value iteration steps are interleaved with approximate value iteration steps in which the new value function is an upper bound to the previously computed value function. This results in a speedup of the total algorithm, however, linear programming is again needed in order to ensure that the new value function is an upper bound to the previous one over the complete belief simplex. In general, computing exact solutions for POMDPs is an intractable problem [20, 16], calling for approximate solution techniques [15, 12].

In practical tasks one would like to compute solutions only for those parts of the belief simplex that are reachable, i.e., that can be actually encountered by interacting with the environment.

This has recently motivated the use of approximate solution techniques for POMDPs which focus on the use of a sampled set of *belief points* on which planning is performed [12, 22, 27, 21, 37, 33], a possibility already mentioned in [15]. The idea is that instead of planning over the complete belief space of the agent (which is intractable for large state spaces), planning is carried out only on a limited set of prototype beliefs that have been sampled by letting the agent interact (randomly) with the environment. PBVI [21], for instance, builds successive estimates of the value function by updating the value and its gradient only at the points of a (dynamically growing) belief set.

In this work we describe PERSEUS, a randomized point-based value iteration algorithm for POMDPs [37, 33]. PERSEUS operates on a large set of beliefs which are gathered by simulating random interactions of the agent with the POMDP environment. On this belief set a number of backup stages are performed. The algorithm ensures that in each backup stage the value of all points in the belief set is improved (or at least does not decrease). Contrary to other point-based methods, PERSEUS backs up only a (random) subset of belief points—the key observation is that a single backup may improve the value of many points in the set. This allows us to compute value functions that consist of only a small number of vectors (relative to the belief set size), leading to significant speedups. We evaluate the performance of PERSEUS on benchmark problems from literature, and it turns out to be very competitive to state-of-the-art methods in terms of solution quality and computation time.

We extend PERSEUS to compute plans for agents which have a continuous set of actions at their disposal. Examples include navigating to an arbitrary location, or rotating a pan-and-tilt camera at any desired angle. Almost all work on POMDP solution techniques targets discrete action spaces; an exception is the application of a particle filter to a continuous state and action space [35]. We report on experiments in an abstract active localization domain in which an agent can control its range sensors to influence its localization estimate, and on results from a navigation task involving a mobile robot with omnidirectional vision in a perceptually aliased office environment.

The rest of the paper is structured as follows: in Section 2 we review the POMDP framework from an AI perspective. We discuss exact methods for solving POMDPs and their tractability problems. Next we outline a class of approximate value iteration algorithms, the so called point-based techniques. In Section 3 we describe and discuss the PERSEUS algorithm, as well as the extension to continuous action spaces. Related work on approximate techniques for POMDP planning is discussed in Section 4. We present experimental results from several problem domains in Section 5. Finally, Section 6 wraps up with some conclusions.

2 Partially observable Markov decision processes

A partially observable Markov decision process (POMDP) models the repeated interaction of an agent with a stochastic environment, parts of which are hidden from the agent's view. The agent's goal is to perform a task by choosing actions which fulfill the task best. Put otherwise, the agent has to compute a plan that optimizes the given performance measure. We assume that time is discretized in time steps of equal length, and at the start of each step the agent has to execute an action. At each time step the agent also receives a scalar reward from the environment, and the performance measure directs the agent to maximize the cumulative reward it can gather. The reward signal allows one to define a task for the agent, e.g., one can give the agent a large positive reward when it accomplishes a certain goal and a small negative reward for each action leading up to it. In this way the agent is steered toward finding the plan which will let it accomplish its goal as fast as possible.

The POMDP framework models stochastic environments in which an agent is uncertain about the exact effect of executing a certain action. This uncertainty is captured by a probabilistic transition model as is the case in a fully observable Markov decision process (MDP) [34, 6]. An MDP defines a transition model which specifies the probabilistic effect of how each action changes the state. Extending the MDP setting, a POMDP also deals with uncertainty resulting from the agent's imperfect sensors. It allows for planning in environments which are only partially observable to the agent, i.e., environments in which the agent cannot determine with full certainty the true state of the environment. In general the partial observability stems from two sources: (1) multiple states lead to the same sensor reading, in case the agent can only sense a limited part of the environment, and (2) its sensor readings are noisy: observing the same state can result in different sensor readings. The partially observability can lead to "perceptual aliasing": the problem that different parts of the environment appear similar to the agent's sensor system, but require different actions. The POMDP represents the partial observability by a probabilistic observation model, which relates possible observations to states.

More formally, a POMDP assumes that at any time step the environment is in a state $s \in S$, the agent takes an action $a \in A$ and receives a reward $r(s, a)$ from the environment as a result of this action, while the environment switches to a new state s' according to a known stochastic transition model $p(s'|s, a)$. The Markov property entails that s' only depends on the previous state s and the action a. The agent then perceives an observation $o \in O$, that may be conditional on its action, which provides information about the state s' through a known stochastic observation model $p(o|s, a)$. All sets S, O, and A are assumed discrete and finite here (but we will generalize to continuous A in Section 3.3).

In order for an agent to choose its actions successfully in partially observable environments some form of memory is needed, as the observations the agent receives do not provide an unique identification of s. Given the transition and observation model the POMDP can be transformed to a belief-state MDP: the agent summarizes all information about its past using a belief vector $b(s)$. The belief b is a probability distribution over S, which forms a Markovian signal for the planning task [3]. All beliefs are contained in the simplex Δ, which means we can represent a belief using $|S| - 1$ numbers. Each POMDP problem assumes an initial belief b_0, which for instance can be set to a uniform distribution over all states (representing complete ignorance regarding the initial state of the environment). Every time the agent takes an action a and observes o, its belief is updated by Bayes' rule:

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

where $p(o|a, b) = \sum_{s' \in S} p(o|s', a) \sum_{s \in S} p(s'|s, a) b(s)$ is a normalizing constant.

As we discussed above, the goal of the agent is to choose actions which fulfill its task as good as possible, i.e., to compute an optimal plan. Such a plan is called a policy $\pi(b)$ and maps beliefs to actions. Note that, contrary to MDPs, the policy $\pi(b)$ is a function over a continuous set of probability distributions over S. The quality of a policy is rated by a performance measure, i.e., by an optimality criterion. A common criterion is the expected discounted future reward $E[\sum_{t=0}^{\infty} \gamma^t r(s_t, \pi(b_t))]$, where γ is a discount rate, $0 \le \gamma < 1$. The discount rate ensures a finite sum and is usually chosen close to 1. A policy which maximizes the optimality criterion is called an optimal policy π^*; it specifies for each b the optimal action to execute at the current step, assuming the agent will also act optimal at future time steps.

A policy can be defined by a value function V_n which determines the expected amount of future discounted reward $V_n(b)$ the agent can gather in n steps from every belief b. The value function of an optimal policy is characterized by the optimal value function V^* which satisfies the Bellman optimality equation $V^* = HV^*$, or

$$V^*(b) = \max_{a \in A} \Big[\sum_{s \in S} r(s, a) b(s) + \gamma \sum_{o \in O} p(o|a, b) V^*(b_a^o) \Big], \tag{2}$$

with b_a^o given by (1), and H is the Bellman backup operator [4]. When (2) holds for every $b \in \Delta$ we are ensured the solution is optimal.

V^* can be approximated by iterating a number of stages, as we will see in the next section, at each stage considering a step further into the future. For problems with a finite planning horizon V^* will be piecewise linear and convex (PWLC) [30], and for infinite horizon (non-episodic) tasks V^* can be approximated arbitrary well by a PWLC value function. We parametrize such a value function V_n by a finite set of vectors (hyperplanes) $\{\alpha_n^i\}$, $i = 1, \ldots, |V_n|$. Additionally, with each vector an action $a(\alpha_n^i) \in A$ is associated, which is the optimal one to take in the current step. Each vector defines a region in the belief space for which it is the maximizing element of V_n. These regions form a partition of the belief space, induced by the piecewise linearity of the value function. Examples of a value function for a two state POMDP are shown in Fig. 1(a) and 1(d). Given a set of vectors $\{\alpha_n^i\}_{i=1}^{|V_n|}$ in V_n, the value of a belief b is given by

$$V_n(b) = \max_{\{\alpha_n^i\}_i} b \cdot \alpha_n^i, \tag{3}$$

where $(\cdot)$ denotes inner product. The gradient of the value function at b is given by the vector $\alpha_n^b = \arg\max_{\{\alpha_n^i\}_i} b \cdot \alpha_n^i$, and the policy at b is given by $\pi(b) = a(\alpha_n^b)$.

2.1 Exact value iteration

Computing an optimal plan for an agent means solving the POMDP, and a classical method for POMDP solving is value iteration. In the POMDP framework, value iteration involves approximating V^* by applying the exact dynamic programming operator H above, or some approximate operator $\tilde{H}$, to an initially piecewise linear and convex value function V_0. For H, and for many commonly used $\tilde{H}$, the produced intermediate estimates $V_1, V_2, \ldots$ will also be piecewise linear and convex. The main idea behind many value iteration algorithms for POMDPs is that for a given value function V_n and a particular belief point b we can easily compute the vector α_{n+1}^b of HV_n such that

$$\alpha_{n+1}^b = \arg\max_{\{\alpha_{n+1}^i\}_i} b \cdot \alpha_{n+1}^i \tag{4}$$

where $\{\alpha_{n+1}^i\}_{i=1}^{|HV_n|}$ is the (unknown) set of vectors for HV_n. We will denote this operation $\alpha_{n+1}^b = \texttt{backup}(b)$. It 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. In particular, defining $r_a(s) = r(s,a)$ and using (1), (2), and (3) we have:

$$\begin{aligned}
V_{n+1}(b) &= \max_a \Big[b \cdot r_a + \gamma \sum_o p(o|a,b) V_n(b_a^o) \Big] &(5)\\
&= \max_a \Big[b \cdot r_a + \gamma \sum_o p(o|a,b) \max_{\{\alpha_n^i\}_i} \sum_{s'} b_a^o(s') \alpha_n^i(s') \Big] &(6)\\
&= \max_a \Big[b \cdot r_a + \gamma \sum_o \max_{\{\alpha_n^i\}_i} \sum_{s'} p(o|s',a) \sum_s p(s'|s,a) b(s) \alpha_n^i(s') \Big] &(7)\\
&= \max_a \Big[b \cdot r_a + \gamma \sum_o \max_{\{g_{a,o}^i\}_i} b \cdot g_{a,o}^i \Big], &(8)
\end{aligned}$$

where

$$g_{a,o}^i(s) = \sum_{s'} p(o|s',a) p(s'|s,a) \alpha_n^i(s'). \tag{9}$$

Applying the identity $\max_j b \cdot \alpha_j = b \cdot \arg\max_j b \cdot \alpha_j$ in (8) twice, we can compute the vector $\texttt{backup}(b)$ as follows:

$$\texttt{backup}(b) = \arg\max_{\{g_a^b\}_{a \in A}} b \cdot g_a^b, \quad \text{where} \tag{10}$$

$$g_a^b = r_a + \gamma \sum_o \arg\max_{\{g_{a,o}^i\}_i} b \cdot g_{a,o}^i. \tag{11}$$

Although computing the vector $\texttt{backup}(b)$ for a given b is straightforward, locating the (minimal) set of points b required to compute *all* vectors $\cup_b \texttt{backup}(b)$ of HV_n is very costly. As each b has a region in the belief space in which its α_n^b is maximal, a family of algorithms tries to identify these regions [32, 11, 13]. The corresponding b of each region is called a “witness” point, as it testifies to the existence of its region. Another set of exact POMDP value iteration algorithms do not focus on searching in the belief space, but instead consider enumerating all possible vectors of HV_n and then pruning useless vectors [18, 10].

As an example of exact value iteration let us consider the most straightforward way of computing HV_n due to Monahan [18]. This involves calculating all possible ways HV_n could be constructed, exploiting the known structure of the value function. We operate independent of a particular b now so (11) can no longer be applied. Instead we have to include all ways of selecting $g_{a,o}^i$ for all o:

$$HV_n = \bigcup_{a,i} \{g_a^i\}, \quad \text{with} \quad \{g_a^i\} = \bigoplus_o \{r_a + \gamma\, g_{a,o}^i\}, \tag{12}$$

where $\bigoplus$ denotes the cross-sum operator.[1] Unfortunately, at each stage a number of vectors exponential in $|O|$ are generated: $|A||V_n|^{|O|}$. The regions of many of the generated vectors will be empty and these vectors as such are useless, but identifying and subsequently pruning them requires linear programming and is therefore costly in high dimensions.

In [38] an alternative approach to exact value iteration was proposed, designed to speed up each exact value iteration step. It turns out that value iteration still converges if exact value update steps are interleaved with approximate update steps in which a new value function V_{n+1} is computed from V_n such that

$$V_n(b) \leq V_{n+1}(b) \leq HV_n(b), \qquad \text{for all } b \in \Delta. \tag{13}$$

This additionally requires that the value function is appropriately initialized, which is trivially realized by choosing V_0 to be a single vector with all its components equal to $\frac{1}{1-\gamma} \min_{s,a} r(s,a)$. Such a vector represents the minimum of cumulative discounted reward obtainable in the POMDP, and is guaranteed to be below V^*. In [38], V_{n+1} is computed by backing up all witness points of V_n for a number of steps. As we saw above, backing up a set of belief points is a relatively cheap operation. Thus, given V_n, a number of vectors of HV_n are created by applying $\texttt{backup}$ to the witness points of V_n, and then a set of linear programs are solved to ensure that $V_{n+1}(b) \geq V_n(b)$, $\forall b \in \Delta$. This is repeated for a number of steps, before an exact value update step takes place. The authors demonstrate experimentally that a combination of approximate and exact backup steps can speed up exact value iteration.

In general, however, computing optimal planning solutions for POMDPs is an intractable problem for any reasonably sized task [20, 16]. This calls for approximate solution techniques. We will describe next a recent line of research on approximate POMDP algorithms which focus on planning on a fixed set of belief points.

[1] Cross-sum of sets $\{R_i\}$ is defined as: $\bigoplus_{i=1}^k R_i = R_1 \oplus R_2 \oplus \ldots \oplus R_k$, with $P \oplus Q = \{\, p+q \mid p \in P,\ q \in Q \,\}$.

2.2 Approximate value iteration

The major cause of intractability of exact POMDP solution methods is their aim of computing the optimal action for every possible belief point in Δ. For instance, if we use (12) we end up with a series of value functions whose size grows exponentially in the planning horizon. A natural way to sidestep this intractability is to settle for computing an approximate solution by considering only a finite set of belief points. The backup stage reduces to applying (10) a fixed number of times, resulting in a small number of vectors (bounded by the size of the belief set). The motivation for using approximate methods is their ability to compute successful policies for much larger problems, which compensates for the loss of optimality.

Such approximate POMDP value iteration methods operating on a fixed set of points are explored in [15] and in subsequent works [12, 22, 21, 37, 33]. In [21] for instance, an approximate backup operator $\tilde{H}_{\mathrm{PBVI}}$ is used instead of H, that computes in each value backup stage the set

$$\tilde{H}_{\mathrm{PBVI}} V_n = \bigcup_{b \in B} \mathtt{backup}(b) \tag{14}$$

using a fixed set of belief points B. The general assumption underlying these so-called *point-based* methods is that by updating not only the value but also its gradient (the α vector) at each $b \in B$, the resulting policy will generalize well and be effective for most beliefs encountered by the agent. Whether or not this assumption is realistic depends on the POMDP's structure and the contents of B, but the intuition is that in many problems the set of 'reachable' beliefs forms a low dimensional manifold in the belief simplex, and thus it can be covered densely enough by a relatively small number of belief points.

Crucial to the control quality of the computed approximate solution is the makeup of B. A number of schemes to build B have been proposed. For instance, one could use a regular grid on the belief simplex, computed, e.g., by Freudenthal triangulation [15]. Other options include taking all extreme points of the belief simplex or use a random grid [12, 22]. An alternative scheme is to include belief points that can be encountered by simulating the POMDP: we can generate trajectories through the belief space by sampling random actions and observations at each time step [15, 12, 22, 21, 37, 33]. This sampling scheme focuses the contents of B to be beliefs that can actually be encountered while experiencing the POMDP model.

The PBVI algorithm [21] is an instance of such a point-based POMDP algorithm. PBVI starts by selecting a small set of beliefs B_0, performs a number of backup stages (14) on B_0, expands B_0 to B_1 by sampling more beliefs, performs again a series of backups, and repeats this process until a satisfactory solution has been found (or the allowed computation time expires). The set B_{t+1} grows by simulating actions for every $b \in B_t$, maintaining only the new belief points that are furthest away from all other points already in B_{t+1}. This scheme is a heuristic to let B_t cover a wide area of the belief space, but comes at a cost as it requires computing distances between all $b \in B_t$. By backing up all $b \in B_t$ the PBVI algorithm generates at each stage approximately $|B_t|$ vectors, which can lead to performance problems in domains requiring large B_t.

In the next section we will present a point-based POMDP value iteration method which does not require backing up all $b \in B$. We compute backups for a subset of B only, but seeing to it that the computed solution will be effective for B. As a result we limit the growth of the number of vectors in the successive value function estimates, leading to significant speedups.

3 Randomized point-based backup stages

We have introduced the POMDP framework which models agents inhabiting stochastic environments that are partially observable to them, and discussed exact and approximate methods for computing successful plans for such agents. Below we describe Perseus, an approximate solution method capable of computing competitive solutions in large POMDP domains.

3.1 Perseus

Perseus is an approximate point-based value iteration algorithm for POMDPs [37, 33]. The value update scheme of Perseus implements a randomized approximate backup operator $\tilde{H}_{\text{Perseus}}$ that improves (instead of maximizes) the value of all belief points in B. Such an operator can be very efficiently implemented in POMDPs given the shape of the value function. The key idea is that in each value backup stage we can improve the value of *all* points in the belief set by only updating the value and its gradient of a randomly selected subset of the points. In each backup stage, given a value function V_n, we compute a value function V_{n+1} that improves the value of all $b \in B$, i.e., we build a value function $V_{n+1} = \tilde{H}_{\text{Perseus}} V_n$ that upper bounds V_n over B (but not necessarily over Δ which would require linear programming):

$$V_n(b) \leq V_{n+1}(b), \qquad \text{for all } b \in B. \tag{15}$$

We first let the agent randomly explore the environment and collect a set B of reachable belief points. We initialize the value function V_0 as a single vector with all its components equal to $\frac{1}{1-\gamma} \min_{s,a} r(s,a)$ as in [38]. Starting with V_0, Perseus performs a number of value function update stages until some convergence criterion is met. Each backup stage is defined as follows, where $\tilde{B}$ is the set of non-improved points:

Perseus randomized backup stage: $V_{n+1} = \tilde{H}_{\text{Perseus}} V_n$

1. Set $V_{n+1} = \emptyset$. Initialize $\tilde{B}$ to B.
2. Sample a belief point b uniformly at random from $\tilde{B}$ and compute $\alpha = \texttt{backup}(b)$.
3. If $b \cdot \alpha \geq V_n(b)$ then add α to V_{n+1}, otherwise add $\alpha' = \arg\max_{\alpha^i \in V_n} b \cdot \alpha^i$ to V_{n+1}.
4. Compute $\tilde{B} = \{b \in B : V_{n+1}(b) < V_n(b)\}$. If $\tilde{B} = \emptyset$ then stop, else go to 2.

Often, a small number of vectors will be sufficient to improve $V_n(b)\ \forall b \in B$, especially in the first steps of value iteration. The idea is to compute these vectors in a randomized greedy manner by sampling from $\tilde{B}$, an increasingly smaller subset of B. We keep track of the set of non-improved points $\tilde{B}$ consisting of those $b \in B$ whose new value $V_{n+1}(b)$ is still lower than $V_n(b)$. At the start of each backup stage, V_{n+1} is set to $\emptyset$ which means $\tilde{B}$ is initialized to B, indicating that all $b \in B$ still need to be improved in this backup stage. As long as $\tilde{B}$ is not empty, we sample a point b from $\tilde{B}$ and compute $\alpha = \texttt{backup}(b)$. If α improves the value of b (i.e., if $b \cdot \alpha \geq V_n(b)$ in step 3), we add α to V_{n+1} and update $V_{n+1}(b)$ for all $b \in B$ by computing their inner product with the new α. The hope is that α improves the value of many other points in B, and all these points are removed from $\tilde{B}$. As long as $\tilde{B}$ is not empty we continue sampling belief points from it and try to add their α vectors.

To ensure termination of each backup stage we have to enforce that $\tilde{B}$ shrinks when adding vectors, i.e., that each α actually improves at least the value of the b that generated it. If not (i.e., $b \cdot \alpha < V_n(b)$ in step 3), we ignore α and insert a copy of the maximizing vector of b from V_n in V_{n+1}. Point b is now considered improved and is removed from $\tilde{B}$ in step 4, together with any other belief points which had the same vector as maximizing one in V_n. This procedure ensures that $\tilde{B}$ shrinks and the backup stage will terminate. A pictorial example of a backup stage is presented in Fig. 1.

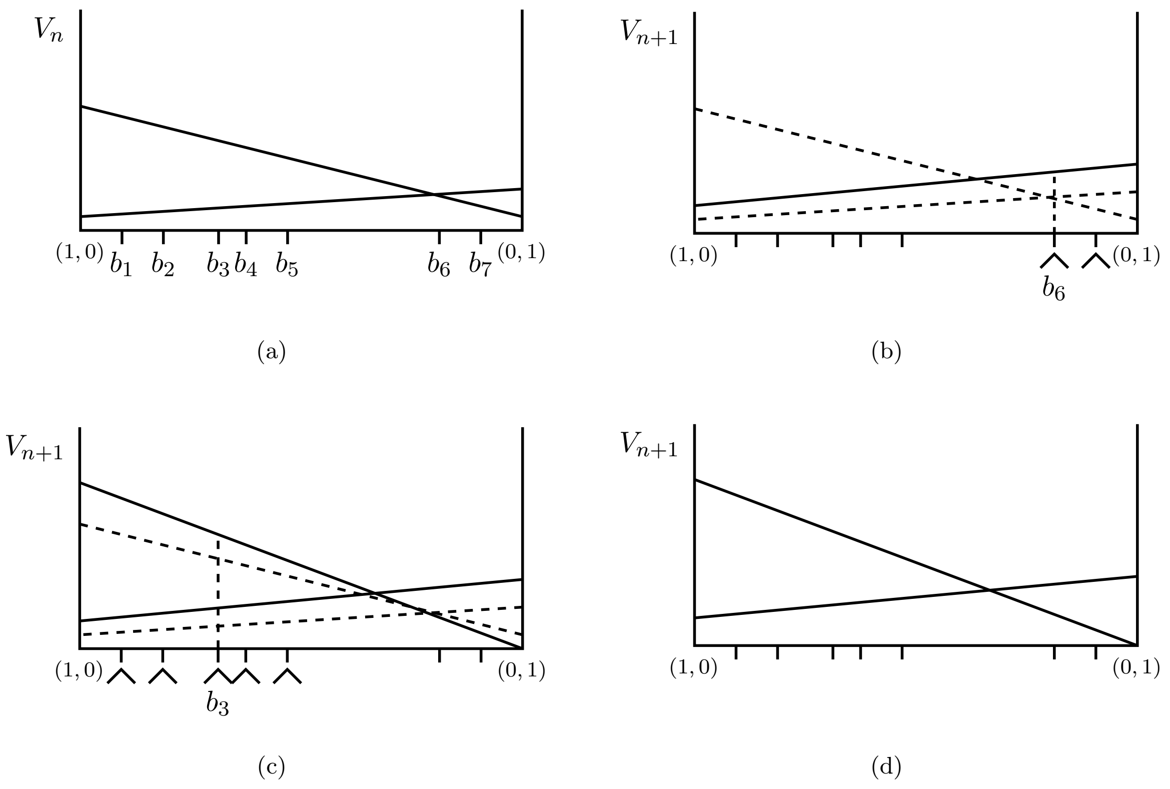


Figure 1: Example of a Perseus backup stage in a two state POMDP. The belief space is depicted on the x-axis and the y-axis represents $V(b)$. Solid lines are α vectors from the current stage and dashed lines are vectors from the previous stage. We operate on a B of 7 beliefs, indicated by the tick marks. The value update stage computing V_{n+1} from V_n proceeds as follows: (a) value function at stage n; (b) start computing V_{n+1} by sampling b_6, add $\alpha = \texttt{backup}(b_6)$ to V_{n+1} which improves the value of b_6 and b_7; (c) sample b_3 from $\{b_1, \ldots, b_5\}$, add $\texttt{backup}(b_3)$ to V_{n+1} which improves b_1 through b_5; and (d) the value of all $b \in B$ has improved, V_{n+1} is done.

3.2 Discussion

The key observation underlying the Perseus algorithm is that when a belief b is backed up, the resulting vector improves not only $V(b)$ but often also the value of many other belief points in B. This results in value functions with a relatively small number of vectors (as compared, e.g., to [22] or [21]). Experiments show indeed that the number of vectors grows modestly with the number of backup stages ($|V_n| \ll |B|$). In practice this means that we can afford to use a much larger B than other point-based methods, which has a positive effect on the approximation accuracy. Furthermore, compared with, e.g., PBVI, building the set B is cheap as we do not compute distances between all $b \in B$ when adding new points. Moreover, note that duplicate entries in B will only affect the probability that a particular b will be sampled in the value update stages, but not the size of V_n.

Perseus can be viewed as a particular instantiation of asynchronous dynamic programming for point-based POMDP value iteration [5]. In asynchronous dynamic programming algorithms no full sweeps over the state space are made, but the order in which states are backed up is arbitrary. This allows an algorithm to focus on backups which may have a high potential impact, as for instance in the prioritized sweeping algorithm for solving fully observable MDPs [19, 2]. A drawback is that the notion of an exact planning horizon is somewhat lost: in general, after performing n backup stages the computed plan will not be considering n steps into the future, but less. By backing up non-improved belief points asynchronously PERSEUS focuses on interesting regions of the (reachable) belief space, and by sampling at random ensures that eventually all $b \in B$ will be taken into account. As we ensure that the value of a particular belief point never decreases, we are guaranteed that PERSEUS will converge [22, 37]: the proof only requires observing that every added vector is always below V^*. Moreover, as we explained above, PERSEUS can handle large belief sets B, thus obviating the use of dynamic belief point selection strategies like those in [12, 22, 21]. Finally note that PERSEUS has no parameters that require tuning by the user, apart from the belief set size.

3.3 Extension to planning with continuous actions

Almost all work on POMDP solution techniques targets discrete action spaces. An exception is the application of a particle filter to a continuous state and action space [35]. We show here that the PERSEUS scheme of 'only–improve' value updates is also very well suited for handling POMDP problems with continuous action spaces.

Instead of considering a finite and discrete action set A we parameterize the agent's actions on a set of k, problem-specific parameters $\theta = \{\theta_1, \theta_2, \ldots, \theta_k\}$. These parameters are real valued and can for instance denote the angle by which a robot rotates. Computing a policy containing such actions requires modifying the `backup` operator defined in Section 2.1, since A now contains an infinite numbers of actions (and therefore maximization over these is not straightforward). The idea here is that instead of maximizing over all $a \in A$, we sample actions at random from A and check whether one of the resulting vectors improves the value of the corresponding belief point. The `backup` operator as defined in (10) is replaced by a backup operator $\alpha = \texttt{backup}'(b)$:

$$\texttt{backup}'(b) = \arg\max_{\{g_a^b\}_{a \in \tilde{A}}} b \cdot g_a^b, \qquad \text{with } \tilde{A} = \{a_i : a_i \text{ is drawn from } A\}, \tag{16}$$

and g_a^b as defined in (11). We draw at random a set $\tilde{A}$ from the continuous set A, in particular specific θ vectors which define actions and which in turn define the g_a^b vectors. In our experiments we let the `backup'` operator sample one θ uniformly at random, but other, more informed, schemes are also possible (for instance, sampling in the neighborhood of the best known θ of a particular b). We can easily incorporate such a backup operator in PERSEUS modifying its backup stage defined in Section 3.1 as follows: instead of using `backup` in step 2 we compute a vector using `backup'`. Using such a randomized backup operator is justified as we check in step 3 whether the vector generated by the sampled action improves the value of the particular belief point. If not, we keep the old vector with the best known action, sampled in a previous backup stage.

An alternative to sampling for handling continuous action spaces is to discretize the space. A computational advantage of reducing the action space to a set of discrete actions is the fact that when A is small enough, one can cache in advance the transition, observation, and reward models for all $a \in A$. In contrast, when we sample a real-valued action we have to compute these models "on the fly" for the sampled action. For instance, in order to generate the transition model $p(s'|s, a)$ the effect of the sampled action for all states needs to be computed. However, the applicability of discretization to a continuous action space is limited, particularly when considering scalability. The number of discrete actions grows exponentially with k, the number of dimensions of θ. For instance, consider a robotic arm with a large number of joints or, as in the experiments of Section 5.2.1, an agent which can control a number of sensors at the same time: discretization would require a number of bins that is exponential in the number of joints or sensors, respectively. Furthermore, the discretization can lead to worse control performance, as demonstrated in Section 5.2.2. Clearly, working directly with continuous actions allows for more precise control.

4 Related work

In Section 2.2 we reported on a class of approximate solution techniques for POMDPs that focus on computing a value function approximation based on a fixed set of prototype belief points. Here we will broaden the picture to other approximate POMDP solution methods. For a more elaborate overview we refer to [12].

A number of heuristic control strategies have been proposed which build on a solution of the underlying MDP. A well-known technique is Q_{MDP} [14], a simple approximation technique that treats the POMDP as if it were fully observable and solves the MDP, e.g., using value iteration. The resulting $Q(s, a)$ values are used to define a control policy by $\pi(b) = \arg\max_a \sum_s b(s)Q(s, a)$. Q_{MDP} can be very effective in some domains, but the policies it computes will not take informative actions, as the Q_{MDP} solution assumes that any uncertainty regarding the state will disappear after taking one action. As such, Q_{MDP} policies will fail in domains where repeated information gathering is necessary.

One way to sidestep the intractability of exact POMDP value iteration is to grid the belief simplex, either using a fixed grid [15, 7] or a variable grid [8, 39]. For every grid point value backups are performed, but only the value of each grid point is preserved and the gradient is ignored. The value of non-grid points is defined by an interpolation rule. The grid based methods differ mainly on how the grid points are selected and what shape the interpolation function takes. In general, regular grids do not scale well in problems with high dimensionality and non-regular grids suffer from expensive interpolation routines.

An alternative to computing an (approximate) value function is policy search: these methods search for a good policy within a restricted class of controllers. For instance, bounded policy iteration (BPI) searches through the space of bounded-size, stochastic finite state controllers [25, 23]. Options for performing the search include gradient ascent [1, 17] and heuristic methods like stochastic local search [9]. Although policy search methods have been applied successfully, choosing an appropriate policy class is difficult and moreover these methods can suffer from local optima.

Compression techniques can be applied to large POMDPs to reduce the dimensionality of the belief space, facilitating the computation of an approximate solution. In [28], Exponential family PCA is applied to a sample set of beliefs to find a low-dimensional representation, based on which an approximate solution is sought. Such a non-linear compression can be very effective, but requires learning a reward and transition model in the reduced space. After such a model is learned, one can compute an approximate solution for the original POMDP using, e.g., MDP value iteration. Alternatively linear compression techniques can be used which preserve the shape of value function [24]. Such a property is desirable as it allows one to exploit the existing POMDP machinery. For instance, in [26] such a compressed POMDP is used as input for BPI, and in [23] linear compression has been applied as a preprocessing step for Perseus.

Little work has been done on the topic of planning with continuous actions in POMDPs. We are only aware of [35] in which particle filters are applied to POMDP domains with a continuous state and action space. As the continuous state space precludes the computation of a traditional belief state, many nice properties (e.g., known shape of the value function) are lost. The belief states are approximated by sets of weighted sample points drawn from the belief distribution. A set of such belief states B is maintained and the value function is represented by $Q(b, a)$ value at each $b \in B$ and updated by value iteration. Nearest neighbor techniques are applied to obtain Q values for beliefs not in B.

HSVI is an approximate value iteration technique which maintains upper and lower bounds to the optimal value function [31]. It performs a heuristic search through the belief space for beliefs at which to update the bounds. Unfortunately, costly linear programming is necessary to compute the upper bound.

Name	$\|S\|$	$\|O\|$	$\|A\|$	$\|B\|$
Tiger-grid	33	17	5	10^3
Hallway	57	21	5	10^3
Hallway2	89	17	5	10^3
Tag	870	30	5	10^4
ALH	100	16	∞	10^4
cTRC	200	10	∞	10^4

(a) Problem domains.

str.	θ_m	θ_n	θ_e	θ_s	θ_w
0	$\{n, e, s, w\}$	0.25	0.25	0.25	0.25
1	$\{n, e, s, w\}$	$[0, 2]$	0.25	0.25	0.25
2	$\{n, e, s, w\}$	$[0, 2]$	$[0, 2]$	0.25	0.25
3	$\{n, e, s, w\}$	$[0, 2]$	$[0, 2]$	$[0, 2]$	0.25
4	$\{n, e, s, w\}$	$[0, 2]$	$[0, 2]$	$[0, 2]$	$[0, 2]$

(b) Action sampling strategies (str) tested in the ALH domain.

Table 1: Experimental setup: (a) problem characteristics; (b) details of the ALH domain.

Finally, an interesting recent work involves combining point-based POMDP techniques with Gaussian Processes to monitor the uncertainty in the consecutive value function estimates [36].

5 Experiments

We will show some experimental results applying Perseus on benchmark problems from the POMDP literature, and present two new POMDP domains for testing Perseus in problems with continuous action spaces. Table 1(a) summarizes these domains in terms of the size of S, O and A, and displays the size of the belief set used as input to Perseus. Each belief set was gathered by simulating random interactions of the agent with the POMDP environment. In all domains the discount factor γ was set to 0.95.

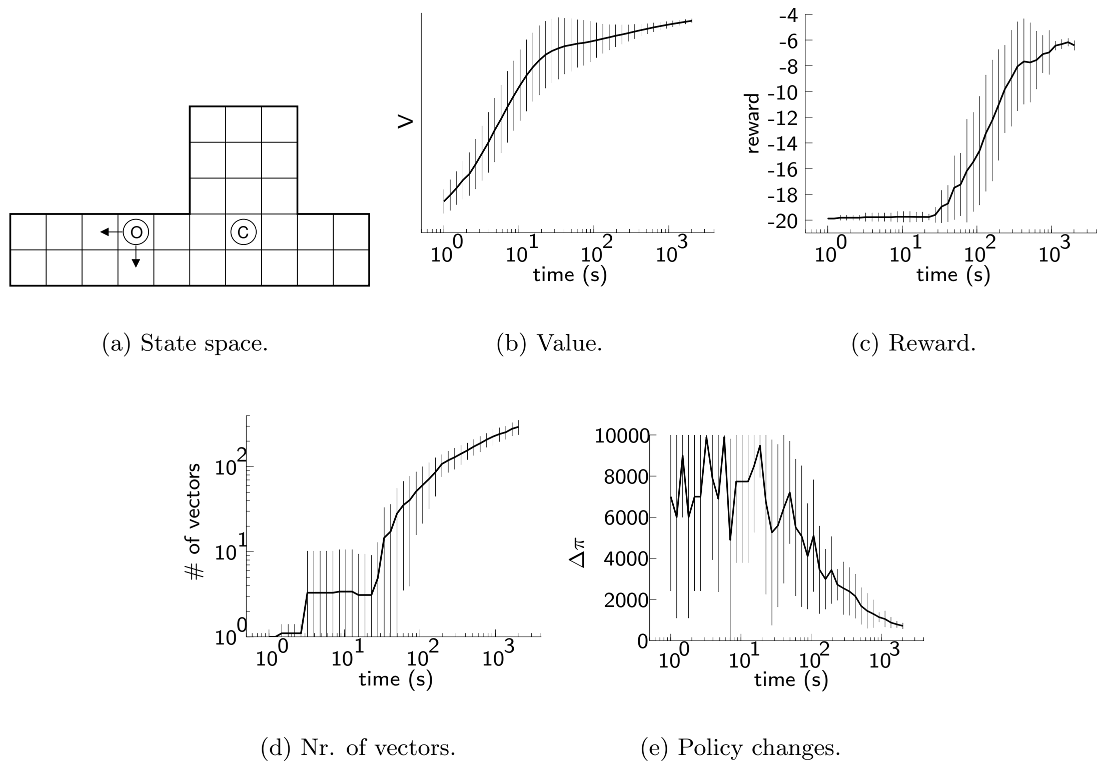
5.1 Discrete action spaces

The Hallway, Hallway2 and Tiger-grid problems (introduced in [14]) are maze domains that have been commonly used to test scalable POMDP solution techniques [14, 8, 39, 21, 31, 37, 33, 23]. The Tag domain [21] is an order of magnitude larger than the first three problems, and is a recent benchmark problem [21, 31, 9, 25, 37, 33, 23].

5.1.1 Benchmark mazes

In [14] three benchmark maze domains were introduced: Tiger-grid, Hallway, and Hallway2. All of them are navigation tasks: the objective is for an agent to reach a designated goal state as quickly as possible. The agent can observe each possible combination of the presence of a wall in four directions plus a unique observation indicating the goal state; in the Hallway problem three other landmarks are also available. At each step the agent can take one out of five actions: {*stay in place, move forward, turn right, turn left, turn around*}. Both the transition and the observation model are very noisy. Table 2(a) through (c) compares the performance of Perseus to other algorithms. The average expected discounted reward R is computed from $1,000$ trajectories for each of the 10 Perseus runs. Each trajectory is sampled by executing the computed policy. We collect the rewards received, discount them and report on the average of these $10,000$ trajectories. Perseus reaches competitive control quality using a small number of vectors and substantially less computation time[2].

[2]Perseus and Q_{MDP} results were computed in Matlab on an Intel Pentium IV 2.4 Ghz; other results were obtained on different platforms, so time comparisons are rough.



(a) State space. (b) Value. (c) Reward.

(d) Nr. of vectors. (e) Policy changes.

Figure 2: Tag: (a) state space with chasing and opponent robot; (b)–(e) performance of PERSEUS.

5.1.2 Tag

The goal in the Tag domain, described in [21], is for a robot to search for a moving opponent robot and tag it. The chasing robot cannot observe the opponent until they occupy the same position, at which time it should execute the *tag* action in order to win the game, and receive a reward of 10. If the opponent is not present at the same location, the reward will be -10, and the robot is penalized with a -1 reward for each motion action it takes. The opponent tries to escape from being tagged by moving away of the chasing robot, it however has a 0.2 chance of remaining at its location. The chasing robot has perfect information regarding its own position and its movement actions {*north, east, south, west*} are deterministic. The state space is represented as the cross-product of the states of the two robots. Both robots can be located in one of the 29 positions depicted in Fig. 2(a), and the opponent can also be in a special *tagged* state, resulting a total 870 states. Tag is a rather large benchmark problem compared to other POMDP problems studied in literature, but it exhibits a sparse structure.

In Fig. 2(b)–(e) we show the performance of PERSEUS averaged over 10 runs, error bars indicate standard deviation within these runs. To evaluate the computed policies we tested each of them on 10 trajectories (of at most 100 steps) times 100 starting positions. Fig. 2(b) displays the value as estimated on B, $\sum_{b \in B} V(b)$; (c) the expected discounted reward averaged over the $1,000$ trajectories; (d) the number of vectors in the value function estimate, $|\{\alpha_n^i\}|$; and (e) the number of policy changes: the number of $b \in B$ which had a different optimal action in V_{n-1} compared to V_n. The latter can be regarded as a measure of convergence for point-based solution methods [15]. We can see that in almost all experiments PERSEUS reaches solutions of virtually equal quality and size.

Table 2(d) compares the performance of PERSEUS with other state-of-the-art methods. The

Tiger-grid	R	$\|\pi\|$	T
HSVI	2.35	4860	10341
PERSEUS	2.34	134	104
PBUA	2.30	660	12116
PBVI	2.25	470	3448
BPI w/b	2.22	120	1000
Grid	0.94	174	n.a.
Q_{MDP}	0.23	n.a.	2.76

(a) Results for Tiger-grid.

Hallway	R	$\|\pi\|$	T
PBVI	0.53	86	288
PBUA	0.53	300	450
HSVI	0.52	1341	10836
PERSEUS	0.51	55	35
BPI w/b	0.51	43	185
Q_{MDP}	0.27	n.a.	1.34

(b) Results for Hallway.

Hallway2	R	$\|\pi\|$	T
PERSEUS	0.35	56	10
HSVI	0.35	1571	10010
PBUA	0.35	1840	27898
PBVI	0.34	95	360
BPI w/b	0.32	60	790
Q_{MDP}	0.09	n.a.	2.23

(c) Results for Hallway2.

Tag	R	$\|\pi\|$	T
PERSEUS	-6.17	280	1670
HSVI	-6.37	1657	10113
BPI w/b	-6.65	17	250
BBSLS	≈ -8.3	30	10^5
BPI n/b	-9.18	940	59772
PBVI	-9.18	1334	180880
Q_{MDP}	-16.9	n.a.	16.1

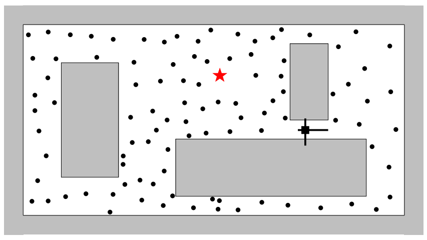
(d) Results for Tag.

Table 2: Experimental comparisons of PERSEUS with other algorithms. PERSEUS results are averaged over 10 runs. Each table lists the method, the average expected discounted reward R, the size of the solution $|\pi|$ (value function or controller size), and the time T (in seconds) used to compute the solution. Sources: PBVI [21], BPI no bias [25], BPI with bias [23], HSVI [31], Grid [8], PBUA [22], and BBSLS [9] (approximate, read from figure).

results show that in the Tag problem PERSEUS displays better control quality than any other method and computes its solution an order of magnitude faster than most other methods. Specifically, its solution computed on 10,000 beliefs consists of only 280 vectors, much less than PBVI which maintains a vector for each of its 1334 $b \in B$. This indicates that the randomized backup stage of PERSEUS is justified: it takes advantage of a large B while the computed value function estimates only grow moderately with the number of backup stages, leading to significant speedups. The controller computed by BBSLS is smaller (30 nodes), but its performance is worse both in control quality and time. It interesting to compare the two variations of BPI, with bias (w/b) or without (n/b). The bias focuses on the reachable belief space by incorporating the initial belief which dramatically increases its performance in solution size and computation time, but it does not reach the control quality of PERSEUS.

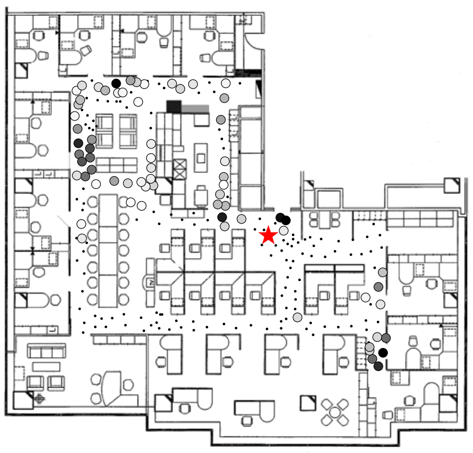
5.2 Continuous action spaces

We applied PERSEUS with continuous actions in two domains: an abstract active localization domain in which an agent can control its range sensors to influence its localization estimate, and a navigation task involving a mobile robot with omnidirectional vision in a perceptually aliased office environment.



(a) ALH: state space.

(b) cTRC: example observation.



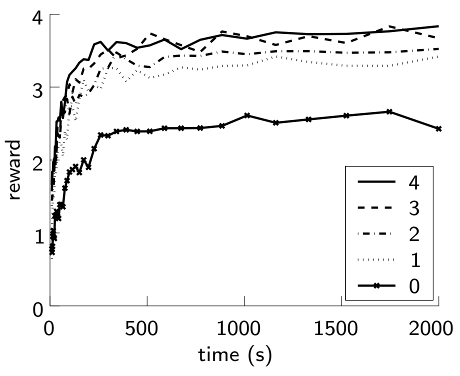
(c) cTRC: environment.

Figure 3: Continuous action space domains: the points indicate the states, ★ depicts the goal state. (a) Environment of the ALH problem: the black square represents the agent. The four lines show the range of its sensors when they are set to $\theta_{n,e,s,w} = \{0.61, 1.12, 0.81, 0.39\}$. (b) cTRC Problem: panoramic image corresponding to a prototype feature vector $o_k \in O$, and (c) its induced $p(s|o_k)$. The darker the dot, the higher the probability.

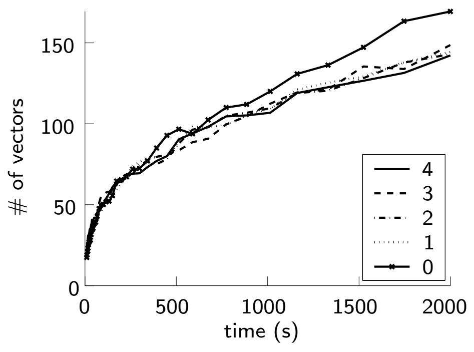
5.2.1 Active localization

We first tested our approach on a navigation task in a simulated environment. The Active Localization Hallway (ALH) environment represents a 20×10 m hallway which is highly perceptually aliased (see Fig. 3(a)). The agent inhabiting the hallway is equipped with four range sensors, each observing one compass direction. The agent can set the range of each sensor, up to a certain limit. We assume a sensor can only detect whether there is a wall within its range or not (but with perfect certainty). The task is to reach a goal location located in an open area where there are no walls near enough for the agent to detect. We would like the agent also to take into account its energy consumption. Moving as well as using the sensor above its default range requires energy and is penalized. The agent is initialized at a random state in the hallway. By moving through the hallway and adjusting its sensors at each step the agent receives information indicating its location. The better it controls the range of its sensors, the more accurate it can localize itself and easier it is to find a path to the goal. Thus, the agent should not only learn what movement actions to take in order to reach the goal, but also how to set its sensors.

The agent's actions are defined by the parameters $\theta = \{\theta_m, \theta_n, \theta_e, \theta_s, \theta_w\}$. At each time step the agent has to set θ_m to one out of four basic motion commands $\{$*north, east, south, west*$\}$ which transports it according to a Gaussian distribution centered on the expected resulting position (translated one meter in the corresponding direction). It sets the range of each of its sensors $\{\theta_n, \theta_e, \theta_s, \theta_w\}$ to a real value in the interval $[0, 2]$ m. We assume that setting a sensor's range higher than its default of 0.5 m costs energy and we penalize with a reward of -0.01 per meter, resulting in a reward of -0.06 if all sensors are fired at maximum range. Each movement is also penalized, with a reward of -0.12 per step. The reward obtainable at the goal location is 10. As Perseus assumes a finite and discrete set S we need to discretize the state space, which is defined as the set of all possible locations of the agent. For discretizing the positions in the map of the environment we performed a straightforward k-means clustering on a random subset



(a) Reward.



(b) Nr. of vectors.

Figure 4: Performance in ALH domain, averaged over 5 runs.

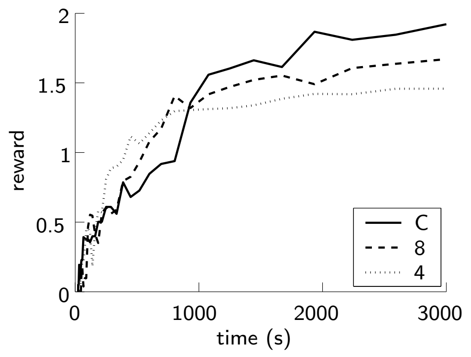
of all possible locations, resulting in a grid of 100 positions, depicted in Fig. 3(a).

To test the feasibility of Perseus, i.e., whether it can compute successful policies by sampling actions at random, we ran it with several different action sampling strategies. The strategy determines from what range each parameter in θ is sampled, and here it defines how many sensors the agent can control. Strategy 0 restricts the agent to only setting θ_m, with $\theta_{n,e,s,w}$ fixed at the default range, while strategy 4 allows full control of all sensors. Table 1(b) summarizes the five strategies we tested. Note that strategy 0 in fact reduced the action space to a discrete set of four actions. We ran our algorithm 5 times for each strategy with different random seeds, plots are averaged over these five runs. To evaluate the computed value function estimates we collected rewards by sampling 10 trajectories from 100 random starting locations. Each trajectory was stopped after a maximum of 100 steps (if the agent had not reached the goal by then).

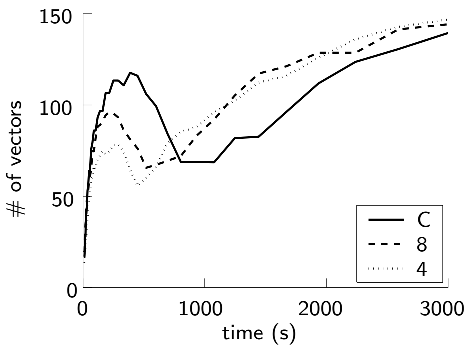
Fig. 4(a) shows the expected discounted cumulative reward for each of the strategies listed in Table 1(b). We see that allowing the agent to control more sensors improves its performance. The algorithm does not seem to be hampered by the increased dimensionality of the action space, as it computes better policies in the same amount of time (using roughly the same amount of vectors). It learns that the advantage of a more accurate localization outweighs the cost of increasing the range of its sensors. We can see that the discrete strategy 0 performs poorly, because of its limited range of sight, even though we can cache its transition, observation and reward models. In Fig. 4(b) we plot the number of vectors in the value function for each strategy. We see that allowing continuous spaces does not result in an excessive size of the value function, its development over time is comparable to the growth in discrete action space problems.

5.2.2 Arbitrary heading navigation

To evaluate Perseus with continuous actions on a more realistic problem and compare against discretized action sampling we also include the cTRC domain. In this problem a mobile robot with omnidirectional vision has to navigate a highly perceptually aliased office environment (see Fig. 3(b) and (c)). It is a variation of the TRC problem introduced in [33], but with a continuous action space. The robot can decide to move 5 meters in an arbitrary direction, i.e., its actions are parameterized by $\theta = \theta_\alpha$ ranging on $[0, 2\pi]$. We assume a Gaussian error on the resulting position.



(a) Reward.



(b) Nr. of vectors.

Figure 5: Performance in cTRC domain, averaged over 5 runs.

For our observation model we used the MEMORABLE[3] robot database that contains a set of approximately 8000 panoramic images collected manually by driving the robot around in a 17 $\times$ 17 meters office environment. As in [33], we compressed the images with PCA and applied k-means clustering to create 10 three-dimensional prototype feature vectors $\{o_1, \ldots, o_{10}\}$. Fig. 3(c) shows the inverse observation model $p(s|o)$ for one observation, and Fig. 3(b) displays the image in the database closest to this particular prototype observation. We used the same technique as in the ALH domain to grid our state space in 200 states (Fig. 3(c)). The task is to reach a certain goal state at which a reward of 10 can be obtained; each action yields a reward of -0.12. Other parameters are the same as in ALH.

We compared our algorithm to two discretized versions of this problem, in which we allowed the robot to sample actions from a set of 4 or 8 headings with equal separation (offset with a random angle to prevent any bias). Fig. 5 displays results for our algorithm, sampling a continuous A ("C") and the two discretized A ("4" and "8"). In particular, Fig. 5(a) shows the superior control quality of the continuous A, accumulating more reward than the discrete cases. Even after 3000s strategy 4 does not reach the goal in 100% of the cases, while the other two strategies do. When employing strategy C, Perseus exploits the ability to move in an arbitrary angle to find a better policy than both discrete cases. Fig. 5(b) plots the number of vectors in the value function for each strategy. Typically this number grows with time, reflecting the need for a more complex plan representation when the planning horizon increases. Interestingly, this figure shows that Perseus can discover that at the same stage, a smaller number of vectors suffice to improve the value of all points in the belief set.

6 Conclusions

The partially observable Markov decision process (POMDP) framework provides an attractive and principled model for sequential decision making under uncertainty. It models the interaction between an agent and the stochastic environment it inhabits. A POMDP assumes that the agent has imperfect information: parts of the environment are hidden from the agent's sensors. The goal is to compute a plan that allows the agent to act optimally given the uncertainty in sensory input and the uncertain effect of executing an action. Unfortunately, the expressiveness of POMDPs is counterbalanced by the intractability of computing exact solutions, which calls for

[3]The MEMORABLE database has been provided by the Tsukuba Research Center in Japan, for the Real World Computing project.

efficient approximate solution techniques. In this work we considered a recent line of research on approximate POMDP algorithms which focus on the use of a sampled set of belief points on which planning is performed.

We presented PERSEUS, a randomized point-based value iteration algorithm for POMDPs. PERSEUS operates on a large belief set sampled by simulating random trajectories through belief space. Approximate value iteration is performed on this belief set by applying a number of backup stages. In each backup stage the algorithm ensures that the value of all points in the belief set is improved (or at least does not decrease). Contrary to other point-based methods, PERSEUS backs up only a (random) subset of belief points. The key idea is that backing up a single belief point can improve the value of many points in the set. Experiments confirm that this allows us to compute value functions that consist of only a small number of vectors (relative to the belief set size), leading to significant speedups. We performed experiments in benchmark problems from literature, and PERSEUS turns out to be very competitive to state-of-the-art methods in terms of solution quality and computation time. We proceeded by extending PERSEUS to compute plans for agents which have a continuous set of actions at their disposal. We demonstrated its viability on two new POMDP problems: an active localization task and a navigation domain in which a robot can move in any direction. To our knowledge, it is one of few POMDP algorithms that handles continuous action spaces.

As future work we would like to move to larger, possibly multiagent domains, by considering alternative state representations. Options include factored states, relational representations, and predictive state representations.

Acknowledgments

We would like to thank Bruno Scherrer, Geoff Gordon, and Pascal Poupart for their comments.

References

[1] D. Aberdeen and J. Baxter. Scaling internal-state policy-gradient methods for POMDPs. In *International Conference on Machine Learning*, Sydney, Australia, 2002.

[2] D. Andre, N. Friedman, and R. Parr. Generalized prioritized sweeping. In *Advances in Neural Information Processing Systems*, Cambridge, MA, 1997. MIT Press.

[3] K. J. Åström. Optimal control of Markov processes with incomplete state information. *Journal of Mathematical Analysis and Applications*, 10:174–205, 2 1965.

[4] R. Bellman. *Dynamic programming*. Princeton University Press, 1957.

[5] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, 1989.

[6] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

[7] B. Bonet. An epsilon-optimal grid-based algorithm for partially observable Markov decision processes. In *Proc. 19th International Conf. on Machine Learning*, pages 51–58, Sydney, Australia, 2002. Morgan Kaufmann.

[8] R. I. Brafman. A heuristic variable grid solution method for POMDPs. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI '97)*, 1997.

[9] D. Braziunas and C. Boutilier. Stochastic local search for POMDP controllers. In *Proc. 19th National Conf. on Artificial Intelligence (AAAI-04)*, San Jose, 2004.

[10] A. R. Cassandra, M. L. Littman, and N. L. Zhang. Incremental pruning: A simple, fast, exact method for partially observable markov decision processes. In *Proc. of Uncertainty in Artificial Intelligence*, Providence, Rhode Island, 1997.

[11] H. T. Cheng. *Algorithms for partially observable Markov decision processes*. PhD thesis, University of British Columbia, 1988.

[12] M. Hauskrecht. Value function approximations for partially observable Markov decision processes. *Journal of Artificial Intelligence Research*, 13:33–95, 2000.

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

[14] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling. Learning policies for partially observable environments: Scaling up. In *Proc. 12th Int. Conf. on Machine Learning*, San Francisco, CA, 1995.

[15] W. S. Lovejoy. Computationally feasible bounds for partially observed Markov decision processes. *Operations Research*, 39(1):162–175, 1991.

[16] O. Madani, S. Hanks, and A. Condon. On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In *Proc. 16th National Conf. on Artificial Intelligence*, Orlando, Florida, July 1999.

[17] N. Meuleau, K.-E. Kim, L. P. Kaelbling, and A. R. Cassandra. Solving POMDPs by searching the space of finite policies. In *Proc. of Uncertainty in Artificial Intelligence*, Stockholm, Sweden, 1999.

[18] G. E. Monahan. A survey of partially observable Markov decision processes: theory, models and algorithms. *Management Science*, 28(1), Jan. 1982.

[19] A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13:103–130, 10 1993.

[20] C. H. Papadimitriou and J. N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of operations research*, 12(3):441–450, 1987.

[21] J. Pineau, G. Gordon, and S. Thrun. Point-based value iteration: An anytime algorithm for POMDPs. In *Proc. Int. Joint Conf. on Artificial Intelligence*, Acapulco, Mexico, Aug. 2003.

[22] K.-M. Poon. A fast heuristic algorithm for decision-theoretic planning. Master's thesis, The Hong-Kong University of Science and Technology, 2001.

[23] P. Poupart. *Exploiting Structure to Efficiently Solve Large Scale Partially Observable Markov Decision Processes*. PhD thesis, University of Toronto, 2004. To appear.

[24] P. Poupart and C. Boutilier. Value-directed compression of POMDPs. In *NIPS 15*. MIT Press, 2003.

[25] P. Poupart and C. Boutilier. Bounded finite state controllers. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, 2004.

[26] P. Poupart and C. Boutilier. VDCBPI: an approximate scalable algorithm for large scale POMDPs. In *NIPS 17*, 2004. To appear.

[27] N. Roy and G. Gordon. Exponential family PCA for belief compression in POMDPs. In *Advances in Neural Information Processing Systems 15*, Cambridge, MA, 2003. MIT Press.

[28] N. Roy, G. Gordon, and S. Thrun. Finding approximate POMDP solutions through belief compression. *Journal of Artificial Intelligence Research*, 2004. To appear.

[29] S. J. Russell and P. Norvig. *Artificial Intelligence: a modern approach*. Prentice Hall, 2nd edition, 2003.

[30] R. D. Smallwood and E. J. Sondik. The optimal control of partially observable Markov decision processes over a finite horizon. *Operations Research*, 21:1071–1088, 1973.

[31] T. Smith and R. Simmons. Heuristic search value iteration for POMDPs. In *Proc. of Uncertainty in Artificial Intelligence*, 2004.

[32] E. J. Sondik. *The optimal control of partially observable Markov decision processes*. PhD thesis, Stanford University, 1971.

[33] M. T. J. Spaan and N. Vlassis. A point-based POMDP algorithm for robot planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2399–2404, New Orleans, Louisiana, 2004.

[34] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[35] S. Thrun. Monte Carlo POMDPs. In S. Solla, T. Leen, and K.-R. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 1064–1070. MIT Press, 2000.

[36] E. Tuttle and Z. Ghahramani. Propagating uncertainty in POMDP value iteration with Gaussian processes. Technical report, Gatsby Computational Neuroscience Unit, Aug. 2004.

[37] N. Vlassis and M. T. J. Spaan. A fast point-based algorithm for POMDPs. In *Benelearn 2004: Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands*, pages 170–176, Brussels, Belgium, Jan. 2004. (Also presented at the NIPS 16 workshop 'Planning for the Real-World', Whistler, Canada, Dec 2003).

[38] N. L. Zhang and W. Zhang. Speeding up the convergence of value iteration in partially observable Markov decision processes. *Journal of Artificial Intelligence Research*, 14:29–51, 2001.

[39] R. Zhou and E. A. Hansen. An improved grid-based approximation algorithm for POMDPs. In *Proc. Int. Joint Conf. on Artificial Intelligence*, Seattle, WA, Aug. 2001.

Acknowledgements

This research is supported by PROGRESS, the embedded systems research program of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW, project AES 5414.

IAS reports