Model-Based Online Learning of POMDPs

Guy Shani, Ronen I. Brafman, and Solomon E. Shimony

Ben-Gurion University, Beer-Sheva, Israel

Abstract. Learning to act in an unknown partially observable domain is a difficult variant of the reinforcement learning paradigm. Research in the area has focused on model-free methods — methods that learn a policy without learning a model of the world. When sensor noise increases, model-free methods provide less accurate policies. The model-based approach — learning a POMDP model of the world, and computing an optimal policy for the learned model — may generate superior results in the presence of sensor noise, but learning and solving a model of the environment is a difficult problem. We have previously shown how such a model can be obtained from the learned policy of model-free methods, but this approach implies a distinction between a learning phase and an acting phase that is undesirable. In this paper we present a novel method for learning a POMDP model online, based on McCallums' Utile Suffix Memory (USM), in conjunction with an approximate policy obtained using an incremental POMDP solver. We show that the incrementally improving policy provides superior results to the original USM algorithm, especially in the presence of increasing sensor and action noise.

1 Introduction

Consider an agent situated in a partially observable domain: It executes an action that may change the state of the world; this change is reflected, in turn, by the agent's sensors; the action may have some associated cost, and the new state may have some associated reward or penalty. Thus, the agent's interaction with this environment is characterized by a sequence of action-observation-reward steps, known as *instances*. In this paper we focus our attention on agents with imperfect and noisy sensors that learn to act in such environments without any prior information about the underlying set of world-states and the world's dynamics, except for information about their sensors' capabilities (namely, a predefined sensor model). This is a known variant of reinforcement learning (RL) in partially observable domains (see, e.g. [3]).

Learning in partially observable domains can take one of two forms; the agent can either learn a policy directly [8, 7], or it can use methods such as the Baum-Welch algorithm for learning HMMs (see, e.g. [2]) to learn a model of the environment, usually represented as a Partially Observable Markov Decision Process (POMDP)[1] , and solve it [4, 9]. This approach has not been favored by researchers, as learning a model appears to be a difficult task, and computing an optimal solution is also difficult.

Moreover, model-free methods naturally support online learning and adapt to changing environments, whereas this is not always the case with model-based methods. In this paper, we return to the model-based approach motivated by a number of recent

[1] See Section 2.1 for an overview of MDPs and POMDPs.

developments: (1) Our recent work on learning POMDP models is able to leverage policies of model-free methods for constructing a POMDP and scales much better than the Baum-Welch algorithm (2) Improvements in approximate POMDP solution methods no longer make it a bottle-neck for this approach. The main contribution of this paper is to show how we can adapt the above ideas to provide online model-based learning of POMDPs, thereby providing a well-rounded approach for model-based learning in partially observable domains. Our algorithm still suffers from the problem that plagues model-free and model-based algorithms for this difficult problem: it works in relatively small domains. However, given the relatively small number of algorithms in this area, it offers a new entry that is both faster, much more robust to sensor noise, and adaptive.

Some research in RL has focused on the problem of *perceptual aliasing* [4], where different actions should be executed in two states where sensors provide the same output. For example, in Figure 1 the left and right corridors are perceptually aliased if sensors can only sense adjacent walls. Model-free methods[2] - such as using variant-length history windows [7] - can be used to disambiguate the perceptually aliased states. When the agent's sensors provide deterministic output, learning to properly identify the underlying world states reduces the problem to a fully observable MDP, making it possible for methods such as Q-learning to compute an optimal policy.

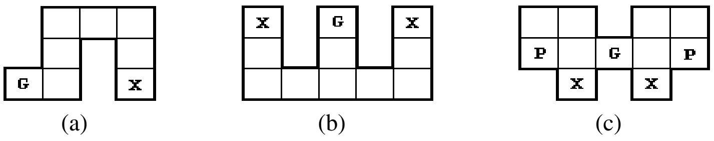


Fig. 1. Three maze domains. The agent receives a reward of 9 upon reaching the goal state (marked with 'G'). Immediately afterwards (in the same transition) the agent is transferred to one of the states marked 'X'. Arrival at a state marked 'P' results in a negative reward of 9.

When sensors provide output that is only slightly noisy, model-free methods produce near-optimal results. However, as noise in the sensors increases, their performance rapidly decreases [10]. This is because disambiguating the perceptually aliased states under noisy sensors does not result in an MDP, but rather in a POMDP. POMDP models are harder to solve, but their solution handles noisy observations optimally.

We have previously shown [11] how well-known model-free methods such as internal memory and McCallum's USM algorithm can be adapted to create after convergence a POMDP model. A solution to such a model provides superior results to the original policy computed by the model-free methods, especially in the presence of noise. This approach, much like earlier methods that rely on the Baum-Welch algorithm, has its disadvantages, as it separates the process into a learning stage and an acting stage. Such separation is undesirable because it is unclear when we should switch from learning to acting, and it also does not handle even slowly changing environments very well.

In this paper we present an algorithm for learning a POMDP model together with its policy online. We adapt the USM algorithm, originally designed for learning a simpler MDP model, for learning the more complicated POMDP model. USM has two

[2] As the discussed problems are properly defined as a POMDP, we call methods that do not learn all the POMDP parameters "model-free", though they may learn state representations.

parts — clustering histories to form a state representation and a planning algorithm based on the learned states. We suggest augmenting the first part to learn a POMDP model using a predefined sensor model, and replacing the original MDP planner with an approximate POMDP planner. The algorithm hence updates the POMDP parameters and continuously computes an approximate policy, using an online version of the Perseus algorithm [12]. Executing policies in a POMDP requires the maintenance of a belief state. The computational cost of our incremental learning algorithm is no greater than the time it takes for the required computation of the belief state. Belief states can also be used to improve the insertion of instances into the POMDP construction algorithm.

Our algorithm makes one important assumption – the existence of a pre-defined sensor model. For instance, in the maze domain, it knows the probability by which a wall is observed given that a wall exists. In general we require a sensor model that provides a distribution over observations given features of a state (rather than given the actual state). While not universally applicable, we believe that such sensor models are quite natural for many domains, robotics in particular.

This paper is structured as follows: we begin (Section 2) with an overview of MDPs, POMDPs and the USM algorithm. We then explain how the USM algorithm can be adapted to incrementally maintain a POMDP in Section 3 and how to compute an approximate policy online in Section 4. We provide an experimental evaluation of our work in Section 5 and conclude in Section 6.

2 Background

2.1 MDPs and POMDPs

A Markov Decision Process (MDP) [5] is a model for sequential stochastic decision problems. An MDP is a four-tuple: $\langle S, A, R, tr \rangle$, where S is the set of the states of the world, A is a set of actions an agent can use, R is a reward function, and tr is the stochastic state-transition function. A solution to an MDP is a policy $\pi : S \rightarrow A$ that defines which action should be executed in each state.

Various exact and approximate algorithms exist for computing an optimal policy, and the best known are policy-iteration [5] and value-iteration [1]. A value function assigns for each state a value $V(s)$ — the expected utility from acting optimally begining in s and on to infinity. Value iteration computes an optimal value function by iteratively solving the equation:

$$V_{n+1}(s) = \max_a R(s,a) + \gamma \sum_{s'} tr(s,a,s')V_n(s') \tag{1}$$

A standard extension to the MDP model is the Partially Observable Markov Decision Process (POMDP) model [3]. A POMDP is a tuple $\langle S, A, R, tr, \Omega, O \rangle$, where S, A, R, tr define an MDP, Ω is a set of possible observations and $O(a, s, o)$ is the probability of executing action a, reaching state s and observing o. The agent is unable to identify the current state and is therefore forced to estimate it given the current observations (e.g. output of the robot sensors) and the agents' history. In many application domains POMDPs are a more precise and natural formalization than an MDP, but using POMDPs increases the difficulty of computing an optimal solution.

History may be represented by a belief state $b(s) = p(s|h)$ — the probability of being in state s after executing and observing history h. The next belief state b_a^o resulting by executing action a and observing o in belief state a can be computed using:

$$b_a^o(s) = \frac{O(a,s,o)\sum_{s'} b(s')tr(s',a,s)}{pr(o|b,a)} \tag{2}$$

2.2 Approximate Solutions to POMDPs

Solving a POMDP is an extremely difficult computational problem, and various attempts have been made to compute approximate solutions that work reasonably well in practice.

Early research [6] suggested the use of an optimal policy for the underlying MDP in conjunction with the belief state to provide a number of approximations that define a policy over belief states.We can select the action that:

– Most Likely State (MLS): corresponds to the maximal Q-value of the state that is most likely given the current belief state.

$$\pi_{MLS}(b) = \text{argmax}_a\, Q(\text{argmax}_s\, b(s), a) \tag{3}$$

– Voting: recommended by most states, weighted by the state probability.

$$\pi_{Voting}(b) = \text{argmax}_a \sum_s b(s)\delta(s,a) \tag{4}$$

where $\delta(s,a) = 1 \Leftrightarrow a = \text{argmax}_{a'}\, Q(s,a)$ and 0 otherwise.

– Q_{MDP}: has the highest Q-value weighted by the state probabilities.

$$\pi_{QMDP}(b) = \text{argmax}_a \sum_s b(s)Q(s,a) \tag{5}$$

An exact solution to a POMDP can be computed using the belief state MDP — an MDP over the belief space of the POMDP. A value function for a POMDP can be described using a set of $|S|$ dimensional vectors defining the expected utility, where each vector $\alpha_a \in V$ corresponds to an action a. We can compute the value function over the belief state MDP iteratively:

$$V_{n+1}(b) = \max_a [b \cdot r_a + \gamma \sum_o pr(o|a,b)V_n(b_a^o)] \tag{6}$$

where $r_a(s) = R(s,a)$ and $\alpha \cdot \beta = \sum_i \alpha(i)\beta(i)$. The computation of the next value function $V_{n+1}(b)$ out of the current one V_n (Equation 6) is known as a *backup* step. Using such a value function V we can define a policy π_V over the belief state:

$$\pi_V(b) = \text{argmax}_{a:\alpha_a \in V}\, \alpha_a \cdot b \tag{7}$$

A point-based algorithm is an algorithm that computes a value function over a finite set of belief points (belief states). Point based algorithms compute an approximate solution as they do not iterate over the entire (infinite) belief space.

Spaan *et al.* explore randomly the world to gather a set B of belief points and then execute the Perseus algorithm (Algorithm 1). Spaan *et al.* also explain how backups can be computed efficiently. Perseus appears to provide good approximations with small sized value functions rapidly.

Algorithm 1. Perseus

```
Input: $B$ — a set of belief points
1: repeat
2:    $\tilde{B} \leftarrow B$
3:    $V' \leftarrow \phi$
4:    while $\tilde{B}$ not empty do
5:       Sample $b \in \tilde{B}$
6:       $\alpha \leftarrow backup(b)$
7:       if $\alpha \cdot b > V(b)$ then
8:          $V' \leftarrow V' \cup \{\alpha\}$
9:       else
10:         $V' \leftarrow V' \cup \{\max_{\beta \in V} \beta \cdot b\}$
11:      $\tilde{B} \leftarrow \{b \in \tilde{B} : V'(b) < V(b)\}$
12:   $V \leftarrow V'$
13: until $V$ has converged
```

2.3 Model Based Approaches

The idea of learning a POMDP model of the environment was examined by early researchers [4, 7] who used a variant of the Baum-Welch algorithm for learning hidden Markov models, refining the state space when it was observed to be inadequate. These methods were slow to converge and could not outperform the rapid convergence and reasonable results generated by model-free methods.

Weirstra and Weiring [13] recently proposed an improvement to McCallums' UDM algorithm allowing it to look farther into the past and speeding its convergence. They however compute an approximate policy using Q-values for the underlying MDP and not by any modern policy computation mechanism. We note that their algorithm is not truly online as it is split into as exploration stage and then a model update stage in order to avoid the long update time of the Baum-Welch algorithm.

Nikovski [9] used McCallum's earlier model-free method, Nearest Sequence Memory (NSM) [7], to identify the states of the world and learn the transition, reward, and observation functions. He showed that the learned models produced superior results to the models obtained by using the Baum-Welch algorithm. His models, however, were tested on domains with little noise, and are much less adequate when sensors are noisy. This is to be expected, as NSM handles noisy environments poorly, where USM can still produce reasonable results, though in no way optimal. Nikovski also did not maintain an incremental model, splitting the learning into a learning phase, followed by model construction and then used the resulting model.

2.4 Utile Suffix Memory

Instance-based state identification [7] resolves perceptual aliasing with variable length short term memory. An instance is a tuple $T_t = \langle T_{t-1}, a_{t-1}, o_t, r_t \rangle$ — the individual observed raw experience. Algorithms of this family keep all the observed raw data (sequences of instances), and use it to identify matching subsequences. It is assumed that two sequences with similar suffixes were likely generated in the same world state.

Utile Suffix Memory creates a tree structure, based on suffix trees for string operations. This tree maintains the raw experiences and identifies matching suffixes. The root of the tree is an unlabeled node, holding all available instances. Each immediate child of the root is labeled with one of the observations encountered during the test. A node holds all the instances $T_t = \langle T_{t-1}, a_{t-1}, o_t, r_t \rangle$ whose final observation o_t matches the node's observation. At the next level, instances are split based on the last action of the instance a_t. We split again based on (the next to last) observation o_{t-1}, etc. All nodes act as buckets, grouping together instances that have matching history suffixes of a certain length. Leaves act as states, holding Q-values and updating them. The deeper a leaf is in the tree, the more history the instances in this leaf share.

The tree is built on-line during the test run. To add a new instance to the tree, we examine its percept, and follow the path to the child node labeled by that percept. We then look at the action before this percept and move to the node labeled by that action, then branch on the percept prior to that action and so forth, until a leaf is reached.

When sensors provide noisy outputs, it is possible that an instance corresponding to a certain location in the world will be inserted into a leaf that represents a different location, due to a noisy observation[10]. Such noisy observations can be reduced by maintaining a belief state. Instead of refereing to a single (possibly noisy) observation, we can consider all possible observations weighted by their probability $p(o|b)$ where b is the current belief state. In USM, $p(o|b)$ is easy to compute as each state (leaf) corresponds to a specific assignment to world features (for example, a specific wall configuration), and therefore $p(o|b) = \sum_{s \in S_o} b(s)$ where S_o is the set of all states that correspond to world configuration o.

We can hence insert a new instance T_t into all states, weighted by $p(o|b_t)$, or replace the noisy observation o_t with the observation with maximal probability $\text{argmax}_o\, p(o|b_t)$. In the experiments reported below we take the second approach.

Leaves should be split if their descendants show a statistical difference in expected future discounted reward associated with the same action. We split a node if knowing where the agent came from helps predict future discounted rewards. Thus, the tree must keep what McCallum calls fringes, i.e., subtrees below the "official" leaves. Figure 2.4 presents an example of a possible USM tree, without fringe nodes.

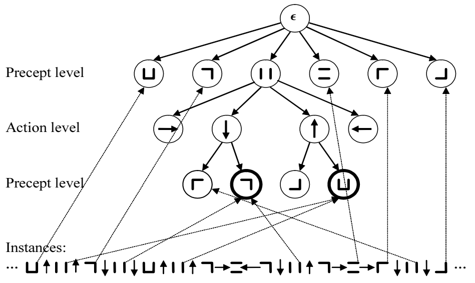


Fig. 2. A possible USM suffix tree generated by the maze in Figure 1. Below is a sequence of instances demonstrating how some instances are clustered into the tree leaves. The two bolded leaves correspond to the same state — the right perceptually aliased corridor. During most executions under deterministic sensor output the above tree structure was generated.

After inserting new instances into the tree, we update Q-values in the leaves using:

$$R(s,a) = \frac{\sum_{T_i \in T(s,a)} r_i}{|T(s,a)|} \tag{8}$$

$$Pr(s'|s,a) = \frac{|\forall T_i \in T(s,a), L(T_{i+1}) = s'|}{|T(s,a)|} \tag{9}$$

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} Pr(s'|s,a)U(s') \tag{10}$$

where $L(T_i)$ is the leaf associated with instance T_i and $U(s) = max_a(Q(s,a))$. We use s and s' to denote the leaves of the tree, as in an optimal tree configuration for a problem the leaves of the tree define the states of the underlying MDP. The above equations correspond to a single step of the value iteration algorithm (see Section 2.1).

Now that the Q-values have been updated, the agent chooses the next action to perform based on the Q-values in the leaf corresponding to the current instance T_t:

$$a_{t+1} = argmax_a Q(L(T_t), a) \tag{11}$$

McCallum uses the fringes of the tree for a smart exploration strategy. In our implementation we use a simple ϵ-greedy technique for exploration.

We note that if a perceptually aliased state can be reached from two different locations, it may have two different leaves that represent it. For example, consider the two leaves in thick line-style in Figure 2.4, corresponding to arriving at the right corridor from above or from below. This phenomenon gives rise to two problems: relevant information is split between leaves, thus requiring a longer learning process, and more seriously, this can lead to a non-compact state space. This is a fundamental problem with USM, and future research should focus on better structures that avoid this duplication, such as using a DAG instead of a tree structure. We note that given any such improvement to USM our algorithms can be modified accordingly.

3 Constructing a POMDP Model over Utile Suffix Memory

Obtaining the POMDP parameters from the USM tree structure is straightforward. The state space (S) is defined as the set of (constantly expanding) tree leaves computed by USM. The actions (A) and observations (Ω) are known to the agent prior to learning the model. The transition function (tr) is defined by Equation 9 and the reward function (R) by Equation 8, as in the original USM. These functions are refined throughout the learning process.

Learning the observation function is harder, as in USM a state always corresponds to a single "true" observation, and all instances mapped to the state hence observe the same sensor output. This "true" state observation is defined by the topmost node below the root, on the path to the state leaf, corresponding to the latest observation in every instance that was added to the leaf. It is therefore unclear how to learn $pr(o|a,s)$ — the probability of observing o **after** reaching state s with action a. We are able to learn a different probability function — the probability of observing o after executing action a **from** state s, but in most of the domains modelled by POMDPs the observation depends on the target state, not on the source state, making the latter definition improper.

It is, however, possible to measure the accuracy of sensors offline, prior to the learning process. For example, we can place a robot in front of an obstacle and measure how likely are its sensors to identify the obstacle. Similarly, it is possible to measure the temperature of a patient multiple times to obtain an error model of the thermometer.

We therefore adopt the approach taken by Shani et al. [10, 11], where an observation model is assumed, and define the observation function based on the observation model. We assume that the agent has some sensor model defining $pr(o|w)$ — the probability that the agent will observe o in world state w. Note that the requirement of a sensor model (which is sufficient for us) is often weaker than the requirement for an observation function. For instance, in maze domains, different rooms with identical wall configurations correspond to different states. However, we only require the ability to assess the likelihood of a certain wall configuration given the sensor's signals, not of the actual state. Thus, in general, it is possible to define a good observation function based on the state's features (which are uniquely determined by the state: the walls are the features in our experiment), but without knowledge of the actual state space (i.e., which rooms actually exist and where).

4 Online POMDP Policy Computation

The Perseus algorithm (Algorithm 1) is executed using a POMDP and a set of belief points. However, since convergence of the algorithm still takes considerable time, we would like to incrementally improve a value function (and hence, a policy) as we learn and act, without requiring the complete execution of Perseus after each step. Our method is an online version of the Perseus algorithm — an algorithm that receives a single belief point and adjusts the computed value function accordingly.

Algorithm 2 is an adaptation of the original algorithm, using two value functions - the current function V and the next function V'. V' is updated until no change has been noted for a period of time, upon which V' becomes the active function V.

Algorithm 2. Iterative Perseus

```
Input: $b$ — a single belief point
1: if $V'(b) < V(b)$ then
2:     $\alpha \leftarrow backup(b)$
3: if $\alpha \cdot b > V(b)$ then
4:     $V' \leftarrow V' \cup \{\alpha\}$
5: else
6:     $V' \leftarrow V' \cup \{\max_{\beta \in V} \beta \cdot b\}$
7: if $V'$ has not been updated in a long while then
8:     $V \leftarrow V'$
9:     $V' \leftarrow \phi$
```

In the original, offline version of Perseus, belief points for updating are selected randomly. The iterative version we suggest selects the points we update not randomly, but following some track through the environment. If this track is chosen wisely (i.e. using a good exploration policy) we can hope that the points that are updated are ones that improve the solution faster.

Using Perseus in conjunction with a model learning algorithm can be problematic, due to the greedy nature of the algorithm. As the value of the next value function over all (tested) belief points always increases, wrong over-estimates, originating from some unlearned world feature, can be persisted in the value function even though they can not be achieved. Such maxima can be escaped using some randomization technique, such as occasionally removing vectors, or by slowly decaying older vectors. We note this problem, even though it does not manifest in our experiments.

5 Experimental Results

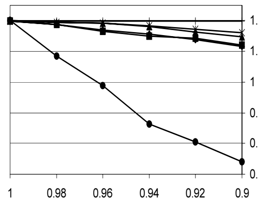
In our experiments we ran the USM-based POMDP on the toy mazes in Figure 1. While these environments are uncomplicated compared to real world problems, they demonstrate important problem features such as multiple perceptual aliasing (Figure 1(b)) and the need for an information gain action (Figure 1(c)). While USM is limited in scaling up to real-world problems, its successor, U-Tree, handles larger domains, and we note that all our methods can be implemented on U-Tree much the same way as for USM.

During execution the model maintains a belief state and states were updated as explained in Section 2.4. The system also ran the iterative Perseus algorithm (Algorithm 2) for each observed belief state. During the learning phase, the next action was selected using the MLS (most likely state) technique (Section 2.1). Once the average reward collected by the algorithms passed a certain threshold, exploration was stopped (as the POMDP policy does not explore).

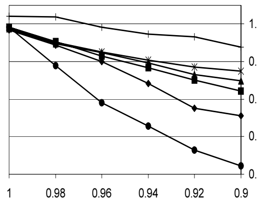
From this point onwards, learning was halted and runs were continued for 5000 iterations for each approximation technique to calculate the average reward gained — MLS, Voting, Q_{MDP} and the policy computed by the iterative Perseus algorithm. In order to provide a gold standard, we manually defined a POMDP model for each of the mazes above, solved it using Perseus and ran the resulting policy for 5000 iterations.

Execution time in our tests was around 6 milliseconds for an iteration of USM, compared to about 234 milliseconds for an iteration of the POMDP learning (including parameter and policy updates), on the maze in Figure 1(b) with sensor accuracy 0.9, on a Pentium 4 with 2.4 GHz CPU and 512 MB memory. The performance of the POMDP learning algorithm is much slower (about n^3) but still feasible for online robotic application, where an action execution is usually measured in seconds. Moreover, much of that time is required for simply updating the belief state, an operation required even for only executing a POMDP policy. For example, the executed policy of the manually defined model (without any learning), takes about 42 milliseconds per iteration.

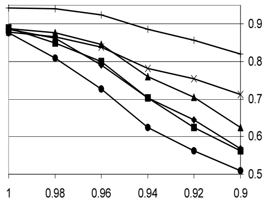
The agent in our experiments has four sensors allowing it to sense an immediate wall above, below, to the left, and to the right of its current location. Sensors have a boolean output with probability p of being correct. The probability of all sensors providing the correct output is therefore p^4. We assume that the agent knows in advance the probability of sensing a wall if a wall exists, and compute the observation function from this information. In the maze there is a single location that grants the agent a reward of 9. In the maze in Figure 1(c) there are two locations where the agent receives a negative reward (punishment) of 9. Upon receiving a reward or punishment, the agent is transformed to any of the states marked by X. If the agent bumps into a wall it pays a cost (a negative reward) of 1. For every move the agent pays a cost of 0.1.



(a) Action noise 0.0.

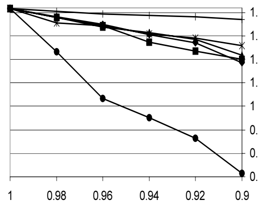


(b) Action noise 0.1.

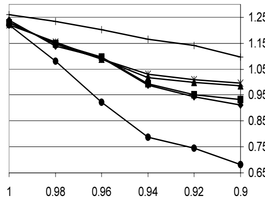


(c) Action noise 0.2.

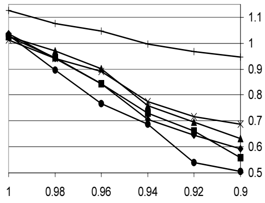
Results for the maze in Figure 1(a).



(d) Action noise 0.0.

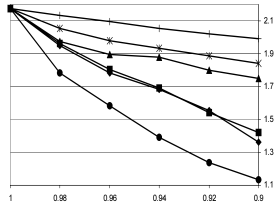


(e) Action noise 0.1.

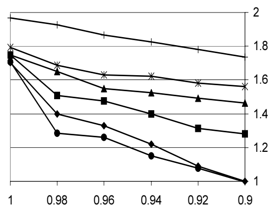


(f) Action noise 0.2.

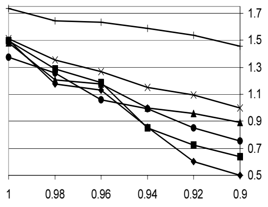
Results for the maze in Figure 1(b).



(g) Action noise 0.0.

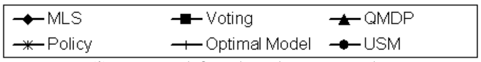


(h) Action noise 0.1.



(i) Action noise 0.2.

Results for the maze in Figure 1(c).



(j) Legend for the above graphs

Fig. 3. Results for the mazes in Figure 1. In all the above graphs, the X axis contains the diminishing sensor accuracy p, and the Y axis marks average reward per agent action. The above results are averaged over 5 different executions for each observation accuracy and method. All variances were below 0.01 and in most cases below 0.005.

Figure 5 presents our experimental results. The graphs compare the performance of the original USM algorithm and our various enhancements: the belief state approximations (MLS, Voting and Q_{MDP}) and the policy computed by the Online Perseus algorithm (Algorithm2), denoted "Policy". We also show the results of the policy for manually defined model, denoted "Optimal Model" as an upper bound.

Observe that the performance of USM decreases sharply as observation noise increases, but the performance of the POMDP based methods remains reasonably high. The improvement is due to the fact that all the POMDP methods model the noise using the belief state, whereas pure USM ignores it. The differences between the POMDP solution methods are not too significant for the first two mazes, and are more noticeable in the last model. Incremental Perseus provides better solutions than the approximations in all the experiments. The third model exhibits more uncertainty in the belief states, and a more pronounced reward variability due to error (this maze is less forgiving w.r.t. deviations from the optimal, especially in the states where the agent observes no walls, where the same action causes a large reward in one state, and a large penalty in the other), making the difference in performance significant. Here, the MDP based methods (MLS, Voting and Q_{MDP}) do not perform nearly as well as the computed policy on that model.

The performance of the policy generated from the USM-based model is not as good as the policy of the manually defined model. This is because the USM-based model has many redundant states, as explained in Section 2.4. The lower performance is not due to the use of the incremental Perseus instead of the offline version. In experiments unreported here, we executed a simulation on a predefined POMDP model, using incremental Perseus to compute a policy. The resulting policy was no worse than the one computed by the offline Perseus on an identical model.

6 Conclusions and Future Work

Model-based algorithms for partially observable environments are widely disfavored due to their slow convergence and the difficulty of computing an optimal policy even when the model is known. This paper presents a model-based algorithm that learns a POMDP model and its solution in conjunction, avoiding the slow computation of the Baum-Welch algorithm. The learned POMDP policy presents superior performance to McCallums' USM in the presence of noisy actions and sensors. The main contribution of this paper is in providing an incremental approach for constructing and solving a POMDP model created online by the agent and demonstrating its effectiveness.

The online Perseus we have presented can also be useful for obtaining policies on standard, predefined, POMDPs and we intend to continue experimenting with it on such domains. Efficient exploration using the online Perseus remains an open question as currently, the policy resulting from it performs very poorly before collecting enough data — much worse than the MDP based approximations.

Improving the construction of the model is probably the main challenge to future work. Currently, the main bottleneck is the size of the learned models. It is possible that USM will create different leaves that correspond to the same state. This leads to large models which require more work to solve and provide lower quality policies. In the future, we plan to examine ways of more aggressively joining states that look similar.

We also believe that model-based methods offer significant advantages in using the current model to guide exploration that is targeted at reaching unknown states and generating instances that improve the model. Indeed, more advanced model-based algorithm may consider issues such as the robustness of the learned model and may attempt to directly model uncertainty about the model parameters, using these to direct additional exploration. Finally, McCallums' USM algorithm provides just one way of constructing a POMDP model, and there may be other methods from which it is easier to induce more accurate models.

Acknowledgments

Partially supported by the Israeli Ministry of Science Infrastructure grant No. 3-942, by the Lynn and William Frankel Center for Computer Sciences, and by the Paul Ivanier Center for Robotics and Production Management at BGU. Guy Shani is partially supported by the Friedman Fund.

References

1. R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1962.
2. J. Bilmes. A gentle tutorial on the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models. Technical Report ICSI-TR-97-021, 1997.
3. A. R. Cassandra, L. P. Kaelbling, and M. L. Littman. Acting optimally in partially observable stochastic domains. In *AAAI'94*, pages 1023–1028, 1994.
4. L. Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *AAAI'02*, pages 183–188, 1992.
5. R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.
6. M. L. Littman, A. R. Cassandra, and L. P. Kaelbling. Learning policies for partially observable environments: Scaling up. In *ICML'95*.
7. A. K. McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, 1996.
8. N. Meuleau, L. Peshkin, K. Kim, and L. P. Kaelbling. Learning finite-state controllers for partially observable environments. In *UAI'99*, pages 427–436, 1999.
9. D. Nikovski. *State-Aggregation Algorithms for Learning Probabilistic Models for Robot Control*. PhD thesis, Carnegie Mellon University, 2002.
10. G. Shani and R. I. Brafman. Resolving perceptual aliasing in the presence of noisy sensors. In *NIPS'17*, 2004.
11. G. Shani, R. I. Brafman, and S. E. Shimony. Partial observability under noisy sensors — from model-free to model-based. In *ICML RRfRL Workshop*, 2005.
12. M. T. J. Spaan and N. Vlassis. Perseus: Randomized point-based value iteration for POMDPs. Technical Report IAS-UVA-04-02, University of Amsterdam, 2004.
13. D. Wierstra and M. Wiering. Utile distinction hidden markov models. In *ICML*, July 2004.