A Survey of Model-Based and Model-Free Methods for Resolving Perceptual Aliasing

Guy Shani
Department of Computer Science
Ben-Gurion University
Beer-Sheva 84105, Israel
shanigu@cs.bgu.ac.il

November 7, 2004

1 Abstract

We focus our attention on agents learning to act in an unknown domain using noisy sensors. Such domains may be modeled by a Partially Observable Markov Decision Process (POMDP) that can be solved optimally. However, when the model of the environment is unknown, most research in the area studies model-free methods — methods that learn to act without learning a model. When the agents' sensors provide deterministic output, model-free methods produce close to optimal results. However, as sensor noise increases, the accuracy of such methods decreases. Another, less explored, option is the model-based approach — learning a POMDP model of the world, and computing an optimal solution using the learned model. In this survey we explore model-based and model-free techniques for handling perceptual aliasing.

2 Introduction

Consider an agent situated in a partially observable domain:

- The agent selects an action from a set of possible actions and executes it. For example, a robot may choose to move to the left, or pick up an item.
- The action may change the state of the world around the agent. The item the robot has picked changes its location, the state of a door may become closed or opened.
- The change is reflected, in turn, by the agent's sensors. After collecting the item the robots sensors indicate that the item is no longer on the floor.
- The action may have some associated cost, and the new state may have some associated reward or penalty. The robot can be penalized for bumping into a wall, or rewarded when delivering an item to its intended destination. The robot movements have a cost in terms of battery charge.

Thus, the agent's interaction with this environment is characterized by a sequence of action-observation-reward steps, known as *instances*(McCallum, 1996). A behavior for such agents is usually specified in the form of a *policy* — a mapping from the world states to actions.

When the agent is given a model of the environment, usually formalized as a Partially Observable Markov Decision Process (POMDP), it can compute an optimal policy. Algorithms

for the computation of an optimal policy were initially considered impractical for reasonably-sized domains, but recent research has focused on exact (e.g. Cassandra, Littman, & Zhang, 1997) and approximate (e.g. Poupart, Boutilier, Schuurmans, & Patrascu, 2002; Braziunas & Boutilier, 2003) algorithms that scale up well.

When the model of the environment is initially unknown, the problem becomes harder to solve. Research has mainly focused on *model-free* techniques — methods that attempt to learn a policy without learning a model of the environment(e.g. McCallum, 1996).

As the agent uses sensors to observe the relevant environmental features, identifying the actual world state presents three possible problems:

- The agent can observes too much data that requires computationally intensive filtering.
- The sensors supply data that is insufficient to identify the current state of the world without taking into consideration past observations as well.
- The agent might be required to operate even though its sensors are insufficient for identifying the current world state.

In this survey we focus only on the second problem — identifying the world state using some form of memory. We also focus on deterministic approaches and do not discuss the possible use of stochastic policies (Jaakkola, Singh, & Jordan, 1995; Williams & Singh, 1998).

The problem of insufficient data leads to a phenomena known as *perceptual aliasing* (Chrisman, 1992), where the same observation is obtained in distinct states where different actions should be performed. Some researchers (e.g Hasinoff, 2002) define all states where the agent observes the same precept to be perceptually aliased, regardless of the optimal action for the state, but we choose to follow Chrisman's definitions. For example, in Figure 1(a) and Figure 1(b) the states marked with X or Y are perceptually aliased. States marked with Z are not perceptually aliased as the optimal action for those states is identical. To compute an optimal policy, the agent must learn to disambiguate the perceptually aliased states. Disambiguating the perceptually aliased states, requires the agent to use some form of internal memory (McCallum, 1996; Peshkin, Meuleau, & Kaelbling, 1999; Meuleau, Peshkin, Kim, & Kaelbling, 1999b; Hayashi & Suematsu, 1999; Suematsu & Hayashi, 1999; Hochreiter & Schmidhuber, 1997; Wiering & Schmidhuber, 1997). We use the term *internal* as we assume that the agent manipulates the memory without any effect from the outside world and without any additional cost.

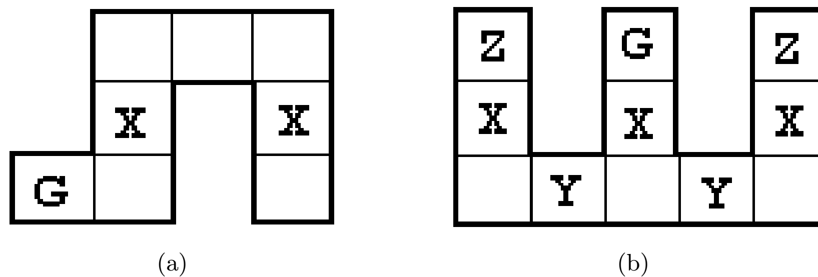

Figure 1: Two maze domains. The agent should navigate to the goal state, marked by G.

The problem of perceptual aliasing is exacerbated when the agent's sensors are not deterministic. For example, if walls can sometimes be reported where none exist, or if a wall is occasionally not detected. Though it is possible to augment some of the robot sensors or add some additional sensors to lower the noise until it can be avoided, in general making observations about the world cannot be expected to be noise-free. Noisy sensors can also be used to

model non robotic problems such as expert systems, such as a system that helps the medical stuff to detect illnesses. For example, a patient with an infection may or may not develop a fever. The observation of a fever is therefore stochastically dependent on whether the patient has an infection or not. The noisy output of the sensors in this example models the probability of having a fever in the presence of an infection, not the thermometer accuracy.

Disambiguating the perceptually aliased states when the environment is observed without noise, reduces the problem to a fully observable Markov Decision Process (MDP), whose states are not necessarily the original world states, but rather the combination of the latest observation and the inner memory state. Applying standard reinforcement learning algorithms for fully observable MDPs such as Q-learning and its extensions provides optimal policies where perceptual aliasing is fully resolved. Most research has therefore focused on disambiguating the perceptual aliasing, and afterwards applying MDP solution methods (e.g. Peshkin et al., 1999).

When sensors become less accurate, resolving the perceptual aliasing is still an important step, yet even if all states are uniquely identified, the noise in the sensors still makes world state identification difficult. When the current state is uncertain, we must apply POMDP solution techniques that take into consideration such problems as the lack of information and the optimal balance between actions that increase the agent's knowledge of the current state and actions that collect rewards.

An alternative to model-free techniques that disregard the noisy sensors is to learn a POMDP model of the environment and afterwards compute an optimal policy based on the learned model. It is possible to apply one of the standard techniques for learning to disambiguate perceptual aliasing, and once it converges, use the states that combine inner memory and observations to define a POMDP model (Nikovski & Nourbakhsh, 2000). An obvious downside of such techniques is the division of the agent life-cycle into learning and acting, which is undesirable. It would be better to incrementally update a POMDP model of the environment, learning to both identify the world states by resolving the perceptual aliasing, and learn an optimal policy and act by it. Such methods, known as *model-based* methods were previously suggested (Chrisman, 1992; McCallum, 1993), but were shown to converge very slowly and were therefore considered impractical.

Although acting in partially observable domains has been the subject of research for more than 40 years, much advancement was made in the past decade. Many researchers try to solve various related problems, from identifying the world through noisy sensors to obtaining an optimal way to behave in the environment. We try to summarize some of these subjects in this survey.

We begin with an overview of fully observable domains, starting with the MDP framework and some basic solution techniques in Section 3, and afterwards discussing model-free (Section 4) and model-based (Section 5) methods in the presence of full observability. In Section 6 we define the POMDP framework, and present some relevant solution techniques for this domain as well. We continue to summarize the main approaches for model-free RL in partially observable domains in Section 7. We provide a more thorough description of McCallum's techniques for instance-based learning in Section 7.3. Section 8 presents the main approaches for model-based techniques for partial observability.

3 MDP

A dynamic system is called Markovian if the system transits to the next state, depending on the current state only. That is, if the system arrives at the same state twice, it will behave the same way (even though the behavior might be stochastic). Therefore, an agent operating in a Markovian system need not use memory at all. It simply observes the current state of the

system to predict the system's future behavior.

In a controlled Markovian system the agent influences the environment through its actions, yet the effect of an action depends solely on the current state. To choose the next action optimally the agent needs only consider the current world state.

Consider for example a robot that transfers packages in a factory. The robot carries a package and should deliver it to a certain location. When the robot arrives at the right location carrying the package, its task is complete and it can return to take a new package. The robot can decide about the next action using its current state — its location, whether it carries a package, the destination of the package, its battery status and so forth — without thinking about its history (how many packages did it already deliver for example).

Such an agent can chose to model the system through a Markov Decision Process (e.g., Howard, 1960; Bellman, 1962; Puterman, 1994; Kaelbling, Littman, & Moore, 1996). Formally, an MDP is a tuple $\langle S, A, tr, R \rangle$ when S is the set of states of the world, A is the set of actions available to the agent, tr is a transition function, and R is a reward function.

3.1 States

A state $s \in S$ is a representation of all the relevant information in the world. Usually it is convenient to describe states as a combination of state variables. In the package delivery problem these variables can be the robot's $\langle x, y \rangle$ location, the current package destination and the battery status. The robot does not need to know the weather outside the factory, or the number of people in China for that matter as this information is irrelevant for the completion of the task. Different values for these variables describe different world states.

We therefore usually define a set of state variables x_i associated with a range of values Y_{x_i} (different variables might have different ranges) and define a state by some assignment of values to the variables $s = (x_0 = y_0, x_1 = y_1, ..., x_n = y_n)$. The number of states is hence exponential in the number of variables.

The size of the state space is a major concern in most applications. Since the solution of the MDP is polynomial in the number of states (not in the number of variables) it is necessary to keep the number of states small.

3.2 Actions

The agent modifies the world by executing actions. An action causes the agent to move from one state to another. For example, the delivery robot actions might be — move forward, turn left, turn right, pick package, drop package and fill battery. It seems that some of these actions only modify the state of the robot, not the world, but as the robot is a part of the description of the world, modifying its state results in modifying the state of the world.

3.3 Rewards

Rewards direct the agent towards desirable states of the world, and keep it away from places it should not visit. The agent's goal is to maximize a stream of incoming rewards, or some function of it, such as the infinite horizon expected discounted reward $\sum_{t=0}^{\infty} \gamma^t r_t$. In our example the robot receives a reward each time it delivers a package, and should be punished (receive a negative reward) each time it delivers a package to the wrong location, or bumps into a wall. Reward is typically not given simply for being in a certain state of the world, but rather for executing some action (such as drop package) in some state. The reward function is therefore written $R(s, a)$. It is also possible to consider stochastic reward functions but in this survey we shall not discuss such definitions.

3.4 Transitions

When the robot tries to move in a certain direction it does not always succeeds. Sometimes it might slip and move left or right. Sometimes the robot tries to turn left, but misses due to some engine inaccuracy or a sudden surge of electricity. The effects of actions are stochastic, leading to a stochastic transition between states. We write $tr(s, a, s')$ for the probability that an agent executing action a at state s will arrive at state s'. Since the model is Markov, the state transition depends only on the current state s and not on the history of the robot.

3.5 Policies and Value Functions

A stationary policy $\pi : S \rightarrow A$ is a mapping from states to actions. It is a convenient way to define the behavior of the agent through stationary policies. Non-stationary policies π_t that provide different actions over different time steps for the same state also exist but are not within the scope of this survey. It was previously observed (Bellman, 1962) that an optimal deterministic policy π^* always exists for any MDP, either the discounted infinite horizon model, the finite horizon model or the average reward model.

An equivalent way to specifying a policy is a value function. A value function $V : S \rightarrow R$ assigns a value for every state s denoting the expected discounted rewards that can be gathered from s. It is possible to define a policy given a value function:

$$\pi(s) = argmax_{a \in A}(R(s, a) + \gamma \sum_{s' \in S} tr(s, a, s')V(s')) \tag{1}$$

The optimal value function is unique and can be defined by:

$$V^*(s) = max_{a \in A}(R(s, a) + \gamma \sum_{s' \in S} tr(s, a, s')V^*(s')) \tag{2}$$

Instead of defining a value for a state we can define a value for a state and an action $Q : SXA \rightarrow R$ known as Q-values, when $Q(s, a)$ specifies the expected future discounted reward from executing action a at state s and acting optimally afterwards.

We can now define an algorithm for computing an MDPs' optimal solution for — value iteration (Algorithm 1) (Bellman, 1962). The algorithm initializes the value function to be the maximal single step reward. On each iteration the value of a state is updated using the last estimation about the value of its successor states. We keep updating state values until they converge.

Algorithm 1 Value Iteration

initialize $V(s) = max_{a \in A} R(s, a)$
while V does not converge **do**
  for all $s \in S$ **do**
    for all $a \in A$ **do**
      $Q(s, a) \leftarrow R(s, a) + \gamma \sum_{s' \in S} tr(s, a, s')V(s')$
    end for
    $V(s) \leftarrow max_{a \in A} Q(s, a)$
  end for
end while
$\forall s \in S, \pi(s) \leftarrow argmax_{a \in A} Q(s, a)$

The stopping criteria for value iteration can be used to create approximate solutions. A typical choice is to stop the algorithm when the maximal difference between two successive

value functions (known as the Bellman residual) drops below ϵ. Value iteration is flexible in that it can be executed asynchronously or in some arbitrary order of assignments for the Q-value and will converge as long as the value of each $Q(s,a)$ gets updated infinitely often. Updating the Q-function based on the transition function is not necessary. We can replace the Q-function update with:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma max_{a' \in A} Q(s',a') - Q(s,a)) \quad (3)$$

As long as the Q-value for each s and a is updated infinitely often, s' is sampled from the $tr(s,a,s')$ distribution, r is sampled with mean $R(s,a)$ and the learning rate α is slowly decreased. Equation 3 is an important foundation for the model-free methods we shall describe shortly.

Other methods for solving MDPs are policy iteration, that searches the space of policies instead of value functions, and linear programming. Linear programming is the only method theoretically known to converge to an optimal solution in polynomial time, but in practice, it is slower than value iteration, policy iteration and their various enhancements.

3.6 RTDP

Real time dynamic programming (RTDP) (Barto, Bradtke, & Singh, 1995; Bonet & Geffner, 2003) is a model solving technique that improves value iteration by trying to learn the optimal policy only for 'relevant' states. RTDP is used only in domains where a set G of goal states and a single start state s_0 exist. The agent receives a reward upon accomplishing the goal but no rewards are received in the process, though they may be negative rewards (costs) for other state action pairs. This is typical for MDPs defined for planning applications. A state s is 'relevant' if an agent starting at s_0 traveling to a goal state $s_g \in G$ using an optimal policy passes through s. RTDP performs a series of simulations of interactions with the environment, starting at s_0 and terminating at a goal state, updating the states the simulation passes using the standard value iteration update rule.

Algorithm 2 RTDP

loop
- $s \leftarrow s_0$
- **while** $s \notin G$ **do**
  - $a \leftarrow \arg\max_{a \in A} Q(s,a)$
  - Update: $Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s' \in S} tr(s,a,s') max_{a' \in A} Q(s',a')$
  - Pick state s' with probability $tr(s,a,s')$
  - $s \leftarrow s'$
- **end while**

end loop

4 Model-free Methods in Markovian Domains

In order to use value or policy iteration, the agent needs a definition of an MDP model $\langle S, A, tr, R \rangle$. While it is reasonable to assume that S — the states of the world and A — the available actions are a part of the problem description, the state transition function and reward function might be initially unknown. Algorithms that learn to act in Markovian domains without a fully specified model of the environment (e.g. R. S. Sutton, 1998) are called model-free methods.

Model-free methods are also popular for solving MDPs with large domains as standard techniques on such domains require a full iteration over all the state space which might be impractical.

4.1 Q-Learning

The most popular method for learning in the absence of a model is Q-learning (Watkins & Dayan, 1992). Q-learning is popular as it is both simple and easy to implement. We define $Q^*(s, a)$ to be the expected discounted reward of executing action a in state s and behaving optimally afterwards. Since $V^*(s) = \max_{a \in A} Q^*(s, a)$ we can write:

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} tr(s, a, s') \max_{a' \in A} Q^*(s', a') \quad (4)$$

In Q-learning (Algorithm 3) the agent executes an action a in state s arriving at state s' with reward r. It then applies Equation 3 to update the Q-value of the former state. If each state-action pair is updated infinitely and the learning parameter α decays slowly, the Q-function converges to Q^*.

Algorithm 3 Q-Learning

initialize $Q(s, a) = 0$
loop
  Choose action a using an ϵ-greedy policy on the Q-values of the current state s
  Execute a, observe reward r and new state s'
  Update: $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma max_{a' \in A} Q(s', a') - Q(s, a))$
  $s \leftarrow s'$
end loop

Choosing the next action should not always select the action with the maximal Q-value to allow for some exploration. A standard exploration technique is e-greedy: choose the action with maximal Q-value with probability $1 - \epsilon$ and choose a random action with probability ϵ. In many cases stopping exploration at any stage is undesirable, since the environment might change and continuing to explore compensates for slow changes.

Note that the rightmost part of the update equation:

$$r + \gamma max_{a' \in A} Q(s', a') - Q(s, a) \quad (5)$$

defines the Bellman residual — the estimated distance from the optimal policy — for the Q-learning algorithm.

4.2 SARSA

A well known variation of Q-learning is the SARSA algorithm (e.g. R. S. Sutton, 1998). SARSA is the same as Q-learning, except that the learning rule (Equation 3) is replaced with:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a)) \quad (6)$$

where s' is the actual next state and a' is the actual action that was selected on the next state[1]. A well known problem with Q-learning performance in online tasks, is that its infinite exploration might cause it to take suboptimal actions, even though the optimal policy has already been discovered. Shutting off exploration is undesirable for several reasons:

[1] Action a' is selected but not executed prior to the update.

- Infinite exploration helps the agent to handle worlds that slowly change over time.
- It is not clear when is it best to stop exploring. Splitting the online process into a learning stage and an acting stage does not ensure convergence to an optimal solution.

While it is usually tolerable to execute suboptimal actions, taking hazardous actions may be intolerable. An agent walking by a cliff, that occasionally takes random actions, might fall down, even though it already knows that falling off the cliff results in heavy damage. An agent aware of its tendency to occasionally execute random actions due to its exploration strategy, will keep a distance from the cliff edge to avoid the penalty that results from falling down due to an exploratory action.

SARSA handles this problem by taking into consideration the exploration policy of the learning algorithm. This will cause the learned policy to keep a distance from dangerous areas. SARSA will learn a policy that is not optimal in the lack of exploration, as it will walk far from danger when walking closer is optimal, but since it suffers less penalties due to exploratory actions, it will accumulate higher rewards.

4.3 Eligibility Traces

Eligibility traces (e.g. Singh & Sutton, 1996) introduce a different approach to the update step in the previous sections. We say that state-action pairs that were recently visited, are more eligible for updating than those that were observed a long time ago. We add an eligibility trace to each pair $\langle s, a \rangle$, initiated when the agent executes action a in state s, decaying exponentially with parameter $0 \leq \lambda \leq 1$ afterwards, i.e. recently visited pairs have a higher trace. When the agent receives a reward or a penalty, the algorithm updates the Q-values of all the states based on their current trace. Therefore all eligible state-action pairs take credit or blame for the event. It is possible to view the trace as a type of local memory, allowing agents to cope with partially observable domains with perceptual aliasing (Loch & Singh, 1998).

Algorithm 4 Sarsa(λ)

initialize $Q(s,a) = 0$
initialize $\eta_0(s,a) = 0$
$t \leftarrow 0$
loop
  Choose action a using an ϵ-greedy policy on the Q-values of the current state s_t
  Execute a_t, observe reward r_t and new state s_{t+1}
  for all state s and action a **do**
    $\eta_t(s,a) = \begin{cases} 1 & , \quad s = s_t, a = a_t \\ \gamma\lambda\eta_{t-1}(s,a) & , \quad otherwise \end{cases}$
    Update: $Q(s,a)_t \leftarrow Q_{t-1}(s,a) + \alpha\eta_t(s,a)(r + \gamma Q(s_t,a_t) - Q(s_{t-1},a_{t-1}))$
  end for
  $t \leftarrow t+1$
end loop

Some well known variations to eligibility traces are the use of replacing traces for instantiating the trace of the current state-action pair $\eta_t(s,a) = 1$ or accumulating traces where $\eta_t(s,a) = \eta_{t-1}(s,a) + 1$ is used. Singh et al. also recommended to set the traces for all other actions from the current state to 0.

$$\eta_t(s,a) = \begin{cases} 1 & , \quad s = s_t, a = a_t \\ 0 & , \quad s = s_t, a \neq a_t \\ \gamma\lambda\eta_{t-1}(s,a) & , \quad otherwise \end{cases} \tag{7}$$

Sarsa(λ) (Algorithm4) is one of the common implementations of reinforcement learning with eligibility traces.

4.4 VAPS

Baird et al. (Baird, 1999; Baird & Moore, 1998) developed an algorithm that can search both in the space of policies and the space of value functions — the Value And Policy Search (VAPS) algorithm. The VAPS algorithm uses gradient descent to minimize some error function on possible policies.

Error functions are defined for instances. An error function should measure the distance between the current policy and the optimal one. An instance T_t represent a suffix of a sequence of interactions of an agent and the environment at time t. We formally define an instance in a fully observable environment (following McCallums' (McCallum, 1996) definitions for an environment with partial observability) as $T_t = \langle T_{t-1}, a_t, r_t, s_t \rangle$ where the agent previously observed instance T_{t-1} and then executed action a_t, received reward r_t and arrived at state s_t. We define $e(T_t)$ — the error at instance T_t. The loss $\varepsilon(T_t)$ induced by a policy that generated the instance T_t is defined by:

$$\varepsilon(T_t) = \sum_{i=0}^{t} e(T_t) \tag{8}$$

and we wish to minimize this loss over all possible instances:

$$B = \sum_{t=0}^{\infty} \sum_{T_t \in \mathcal{T}_t} pr(T_t)\varepsilon(T_t) \tag{9}$$

where $\mathcal{T}_t$ is the set of all possible instances of length t.

For example, we can use the squared Bellman residual for the Q-learning algorithm (Equation 5) as the error measure:

$$e_{QL}(T_t) = \sum_{s \in S} tr(s_{t-1}, a_t, s)(r_{t-1} + \max_{a \in A} \gamma Q(s,a) - Q(s_{t-1}, a_{t-1}))^2 \tag{10}$$

We can define a more sophisticated error measure:

$$e_{SARSA}(T_t) = \sum_{s \in S} tr(s_{t-1}, a_t, s) \sum_{a \in A} pr(a_t = a | s_{t-1})(r_{t-1} + \gamma Q(s,a) - Q(s_{t-1}, a_{t-1}))^2 \tag{11}$$

that averages over the actions the policy might choose from state s_{t-1} instead of using the action that maximizes the Q-values. This error measure is closely related to the SARSA algorithm that takes into account when updating the Q-values the policy that chooses future actions.

Baird et al. also define an error measure for policies (rather than for Q-values) that simply maximizes the discounted utility:

$$e_{policy}(T_t) = b - \gamma^t r_t \tag{12}$$

where b is an constant used to make the error function values positive. It is also possible to linearly combine error measures:

$$e_{SARSA-policy}(T_t) = \beta e_{policy}(T_t) + (1-\beta) e_{SARSA}(T_t) \tag{13}$$

As the error above combines both policy and value search errors it is known as VAPS — Value And Policy Search.

Some stochastic policies can be represented as a vector of weights. For example, a policy that uses a Boltzman distribution on the Q-values to perform exploration can be represented as a table of weights when $w_{s,a} = Q(s,a)$. We can now define (Peshkin et al., 1999) an incremental version of the VAPS algorithm using an exploration trace $ET_{s,a,t}$ used to update the weight (Q-values) table. The VAPS algorithm incrementally updates the exploration trace and weight table after instance T_t was observed:

$$\Delta ET_{s,a,t} = \frac{\partial}{\partial w_{s,a}} \ln pr(a_{t-1}|s_{t-1}) \tag{14}$$

$$\Delta w_{s,a} = -\alpha(\frac{\partial}{\partial w_{s,a}} e(T_t) + e(T_t) ET_{s,a,t}) \tag{15}$$

Baird et al. show that the gradients of the immediate error e with respect to weight $w_{s,a}$ are easy to calculate. The equations above approximate the stochastic gradient descent of the error measure B. Updates to the weights $w_{s,a}$ result in an update to the the stochastic policy represented by $pr(a|s)$.

4.5 Hierarchical approaches

Parr et al. (Parr & Russell, 1997) present the Hierarchy of Abstract Machines (HAM) architecture for reinforcement learning. A HAM is constructed of several layers of abstraction, each defined by a finite state machine whose states are either an action execution or a call to a lower level machine. Parr et al. also show how to learn a HAM representing a policy without learning a model of the environment.

5 Model-based Methods in Markovian Domains

As we previously noted, the transition and reward functions might be initially unknown. The most straight-forward approach to acting in an unknown environment that can be modeled as an MDP is to learn the two functions by roaming the world and keeping statistics of the state transitions and rewards. Once a model has been learned, the agent can use a policy computation algorithm, such as value-iteration, to calculate an optimal policy. There are two major drawbacks to this method:

- Arbitrary division between learning and acting is undesirable. We usually prefer an agent that exhibits a learning curve, so that with each new experience it behaves a little better. One important advantage of model-free methods is that they keep exploring the world, and can therefore adapt to slow changes in the environment. Learning first and acting afterwards does not posses this attractive feature.

- It is unclear how the agent should explore the world. Random walks are undesirable as they might take a long time to learn the interesting parts of the world, while wasting time in places where no goal awaits. Random exploration might also cause an agent to repeatedly visit dangerous sections, receiving unneeded punishments.

The simple approach to model learning, known as the Certainly-Equivalent (e.g. Kaelbling et al., 1996) method, updates the model on each step (or every constant number of steps), solves it obtaining a new policy and then acts by the new policy. While this approach addresses the issues above, it is extremely computationally inefficient, and hence, impractical.

5.1 Dyna

The Dyna system (Sutton, 1991) tries to combine model-free and model-based advantages; It incrementally learns both the Q-values and the model parameters, updating the policy and model simultaneously. Dyna keeps statistics for the state transitions and rewards and uses them to perform a part of the value iteration updates. As we noted before, value iteration does not require the agent to run sequential updates as long as each state-action pair is visited often enough.

Algorithm 5 Dyna(k)

---

initialize $Q(s,a)=0$, $tr(s,a,s')$, $R(s,a)$
loop
  Choose action a using an ϵ-greedy policy on the Q-values of the current state s
  Execute a, observe reward r and new state s'
  Update: $tr(s,a,s')$ and $R(s,a)$ statistics
  Update: $Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s' \in S} tr(s,a,s') max_{a' \in A} Q(s',a')$
  for $i = 0$ to k **do**
    Choose s_i and a_i randomly
    Update: $Q(s_i,a_i) \leftarrow R(s_i,a_i) + \gamma \sum_{s' \in S} tr(s,a,s') max_{a' \in A} Q(s',a')$
  end for
end loop

Prioritized sweeping (Moore & Atkeson, 1993) improves Dyna by using a smarter selection of the k updates. Instead of choosing to update states at random we prioritize the selection of a pair $\langle s,a \rangle$ according to the variance in the latest updates to the successors states $\{s' \in S | tr(s,a,s') > 0\}$. Therefore, when an unexpected reward was observed, the system propagates this information to its predecessor states.

6 POMDP

All previous techniques assume that the agent always has full knowledge about its current state. For example, not only that such agents know their own current location, they are also aware of all other items in the world, regardless of whether they can see them or not. In practice this assumption is unrealistic. For example, robots have sensors such as cameras, proximity detectors, compasses, providing incomplete data over the surrounding. A robot in a maze cannot usually know exactly its $\langle x,y \rangle$ coordinates. It observes walls around it and its direction and has to deduce from this data about its whereabouts. Moreover, sensors are inaccurate and return noisy output, causing more confusion as to the current state of the world.

Formally, we assume that the agent is acting in a Markovian world, but it gains information over the state of the world through observations. The POMDP model (e.g. Sondik, 1971; Poupart, 2002; Kaelbling, Littman, & Cassandra, 1998; Hasinoff, 2002) for describing such agents is a tuple $\langle S, A, tr, R, \Omega, O \rangle$ where S, A, tr and R define an MDP (Section 3), Ω is a set of observations and $O(a,s,o)$ is the probability that an agent will observe $o \in \Omega$ after executing a, reaching state s.

6.1 Solving POMDPs

Even when the POMDP model is fully known, solving it is a difficult task (e.g. Murphy, 2000). A common approach is to define a belief state — a vector $b = \langle b_0, ..., b_{|S|} \rangle$ where b_i is the

probability of the agent being at state s_i. When the agent is at belief state b, executing action a and observing o, it can calculate the new belief state:

$$b_o^a(s) = pr(s|a,o,b) \tag{16}$$
$$= \frac{pr(o|s,a)pr(s|b,a)}{pr(o|b,a)} \tag{17}$$
$$= \frac{O(a,s,o)\sum_{s'\in S} b(s')tr(s',a,s)}{pr(o|b,a)} \tag{18}$$

As we have noted above, we would like to adapt this standard equation to use the observation function $O(s,a,o)$:

$$b_o^a(s) = pr(s|a,o,b) \tag{19}$$
$$= \sum_{s'} pr(s|a,o,b,s')pr(s'|a,o,b) \tag{20}$$
$$= \frac{\sum_{s'} pr(s|a,s')pr(s'|a,b)pr(o|a,s')}{pr(o|b,a)} \tag{21}$$
$$= \frac{\sum_{s'} tr(s',a,s)b(s')O(s',a,o)}{pr(o|b,a)} \tag{22}$$

The state estimator function $SE(b,a,o)$ outputs the new belief state b_o^a given the former belief state b, last action a and current observation o using Equation 16.

Given the belief states, we can define and solve the belief-state MDP:

$$R(b,a) = \sum_{s\in S} b(s)R(s,a) \tag{23}$$
$$tr(b,a,b) = \sum_{o\in\Omega} pr(b'|b,a,o)pr(o|b,a) \tag{24}$$
$$pr(b'|b,a,o) = \begin{cases} 1 & , \quad SE(b,a,o) = b' \\ 0 & , \quad otherwise \end{cases} \tag{25}$$
$$pr(o|b,a) = \sum_{s\in S} b(s) \sum_{s'\in S} b(s)tr(s,a,s')O(a,s',o) \tag{26}$$

We can rewrite Equation 26 into:

$$pr(o|b,a) = \sum_{s\in S} b(s)O(s,a,o) \tag{27}$$

While belief-state MDPs have a continuous state space they have a piecewise linear and convex value function and can be solved by value iteration (Kaelbling et al., 1998) or policy iteration algorithms (Hansen, 1998). In recent years much work was done on POMDP solvers. Problems that were considered too large to solve can now be approximated efficiently (Parr & Russell, 1995; Littman, Cassandra, & Kaelbling, 1995b). For example, Poupart et al. (Poupart et al., 2002) suggest to compress POMDPs with a large state space, using the solution of the compressed model as an approximation of the optimal policy.

6.2 MDP-based Approximations

Prior to the presentation of the witness algorithm (Littman, Cassandra, & Kaelbling, 1995a), as the earlier methods for the solution of the belief-state MDP were considered impractical for

real-world problems, research has suggested a number of approximation methods that, using the Q-function of the underlying MDP and a belief state, define a policy for a POMDP.

The *most likely state* (MLS) method chooses the optimal action for the state with the highest probability in the current belief state. It does not consider the degree to which we are certain of the current state, nor the value of the optimal function. The *Voting* method addresses these concerns by selecting the action that has the highest probability mass in the belief state vector. The Q_{MDP} approximation improves on Voting in that it takes the Q-values into consideration. In Q_{MDP} we select the action that maximizes $\sum_s Q(s,a)b(s)$, where $b(s)$ is the value of the belief vector for state s.

6.3 Incremental Pruning

The belief-state MDP value function can be represented using a set of vectors. As the value function is piecewise linear and convex it can be represented in the form of $|S|$-dimensional vectors defining the upper envelope. Each vector α_a represents the expected value from executing action a in any possible belief state.

The value iteration equation that computes the next value function V_{t+1} given the current value function V_t:

$$V_{t+1}(b) = max_{a\in A}(\sum_{s\in S} b(s)R(s,a) + \gamma \sum_{o\in\Omega} pr(o|b,a)V_t(b_o^a)) \tag{28}$$

can be decomposed into:

$$V_{t+1}(b) = \max_{a\in A} V^a(b) \tag{29}$$

$$V^a(b) = \sum_{o\in\Omega} V_o^a(b) \tag{30}$$

$$V_o^a(b) = \frac{\sum_{s\in S} b(s)R(s,a)}{|\Omega|} + \gamma pr(o|b,a)V_t(b_o^a) \tag{31}$$

where

$$pr(o|a,b) = \sum_{s'\in S} O(a,s',o) \sum_{s\in S} b(s)tr(s,a,s') \tag{32}$$

is the normalizing factor.

The value iteration equations can therefore be written in terms of sets of vectors, where we compute the next set of vectors $\mathcal{V}_{t+1}$ given the former set of vectors $\mathcal{V}_t$:

$$\mathcal{V}_{t+1} = \bigcup_{a\in A} \mathcal{V}^a(b) \tag{33}$$

$$\mathcal{V}^a = \bigoplus_{o\in\Omega} \mathcal{V}_o^a(b) \tag{34}$$

$$\mathcal{V}_o^a = \{\tau(\alpha,a,o)|\alpha \in \mathcal{V}_t\} \tag{35}$$

$$\tau(\alpha,a,o)(s) = \frac{R(s,a)}{|\Omega|} + \gamma \sum_{s'\in S} b(s')tr(s',a,s)O(a,s,o) \tag{36}$$

where $A \oplus B = \{\alpha + \beta | \alpha \in A, \beta \in B\}$.

Cassandra et al.(Cassandra et al., 1997) show that when computing $\mathcal{V}'$, $\mathcal{V}^a$ and $\mathcal{V}_o^a$, the sets of vectors can be pruned to remove dominated elements, keeping the vector sets minimal. The Incremental Pruning (IP) algorithm, using smart pruning techniques, can provide exact solutions to problems with a small state space.

6.4 Finite State Controllers

A Finite State Controller(Hansen, 1998) (also known as a policy graph) is a finite state automaton, where states are labelled by actions, and edges labeled with observations. When the agent is at a certain state in the FSC it executes the action associated with the state. The action triggers a change in the world that the agent may observe through its sensors. Given the observation the agent changes the internal state of the FSC. Such controllers can be used to define a policy for a POMDP.

Hansen show that given a POMDP model we can learn an FSC that captures an optimal policy, and considers this a form of policy iteration for POMDPs. In general, however, the FSC can grow such that the computation of an optimal controller becomes infeasible. FSCs however provide a good method for approximations techniques, by fixing the size of the controller, avoiding the problem above, the agent can find the best controller given the pre-defined size (Meuleau, Kim, Kaelbling, & Cassandra, 1999a; Poupart & Boutilier, 2004; Braziunas & Boutilier, 2003).

7 Model-free Methods for POMDPs

When dealing with a fully observable MDP, we assume that the agent might be unaware of the state transition and reward probabilities. The problem becomes harder in a partially observable domain when the agent might be unaware of the state space at all. The agent is only aware of aspects of the problem that are a part of its structure — the actions it can execute, its sensors, their possible output signals and their accuracy. Model-free methods (e.g. Aberdeen, 2003) try to learn how to act without learning the unknown parameters of the model.

The trivial approach to cope with an unknown state space observed only through sensors is to use the observation space instead of the unknown state space, i.e. assume that each observation corresponds to a single state. Using observations instead of states might lead to two opposite problems:

- The agent might have numerous sensors, each applicable to different aspects of the problem. The Mars Exploration Rover for example can sense many things about its surroundings, most of which are needed for the Mars research, but irrelevant for its navigational mission. Such agents need to learn to distinguish between the relevant and irrelevant observations. We do not handle such problems in this survey.

- The observation space might be smaller than the state space, causing the agent to suffer from the problem of perceptual aliasing (e.g. Chrisman, 1992). Having numerous states correspond to the same precept can be beneficial when the optimal action for each state is the same, as it diminishes the size of the state space. When two states generate the same observation, yet the agent needs to execute different actions in each state, we call the states perceptually aliased, and the agent must learn to differentiate them in order to compute an optimal policy, a problem we focus on below.

7.1 Reactive memoryless policies

As noted above, the simplest approach for handling an unknown state space, is to use the observation space instead of the state space, and then apply some method applicable for fully observable domains such as Q-learning or Sarsa to solve it, computing $Q(o, a)$ rather than $Q(s, a)$. These policies are known as memoryless or reactive, as they react to the latest observation solely without keeping track of past events. Littman (Littman, 1994) provided theoretical

limitations for memoryless policies. Littman also presented a branch-and-bound algorithm for computing optimal memoryless policies and examined its performance on a number of examples. The branch-and-bound algorithm relied on the existence of a POMDP model, making it model-based, rather than model-free.

Whitehead et al. (Whitehead & Ballard, 1991) suggested in their Lion algorithm to decrease the utility of states that are perceptually aliased so that the agent will try to avoid these states if possible. While this approach can be effective in some domains, it is generally required to be able to somehow visit perceptually aliased states.

Jaakkola et al. (Jaakkola et al., 1995) showed that a deterministic reactive policy can perform arbitrarily worse than stochastic policies. Their work was continued by Williams et al. (Williams & Singh, 1998) who implemented an online version of the stochastic algorithm and tested it. They showed their algorithm to converge on a problem, but with inferior results (about 40% of the average reward per step) to Parr's model based approximation techniques (Parr & Russell, 1995) and the Witness algorithm.

As we have noted above, it is possible to view eligibility traces as a type of short-term memory, since they update the latest state-action trajectory of the agent, and can therefore be used (Loch & Singh, 1998) to augment the ability to handle partial observability. Loch et al. explored problems where a memoryless optimal policy exists and demonstrated that Sarsa(λ) (Algorithm 4) can learn an optimal policy for such domains.

Baird et al. (Baird, 1999; Baird & Moore, 1998) show that the VAPS algorithm (see Section 4.4 for details) will converge for environments with partial observability where a memoryless policy exists. They did not however show VAPS to work better than other algorithms, neither in the presence of full nor partial observability.

Hayashi et al. (Hayashi & Suematsu, 1999) adapt classifier systems — rule based systems that learn the rules online — to partial observability. They implemented a generalization of a classifier system and experimented with it, showing it to converge on a number of problems where a memoryless policy can be calculated. They did not compare the performance of their algorithms to any other model-free technique.

7.2 Memory based approaches

Early researchers already noticed that the problem of perceptual aliasing in general could not be optimally solved by reactive policies. Researchers therefore begun to study methods that augmented the observations with some kind of internal memory. We use the term internal for any mechanism that the agent can modify directly, naming any other part of the environment, modified through the agents actions, external. The following sections introduce some of the main approaches for using internal memory to solve the problem of perceptual aliasing. Figure 7.2 shows an agent augmented with an internal memory module.

It is important to note that when a POMDP has no noisy sensor or very little noise, disambiguating the perceptual aliased states, causes the POMDP to reduce to an MDP. Since all strategies to obtain a good policy from the learned model described below originate from the MDP domain (such as Q-learning and its variations), they perform well once the POMDP was reduced to an MDP. However, when the sensors provide highly noisy output, using those algorithms may produce suboptimal policies.

7.2.1 Finite size histories

Researchers advocating the use of memoryless policies suggested to use the last k observations instead of the immediate observation solely. The size of the history window in such cases is fixed for all observation sequences, given as a parameter rather than computed online.

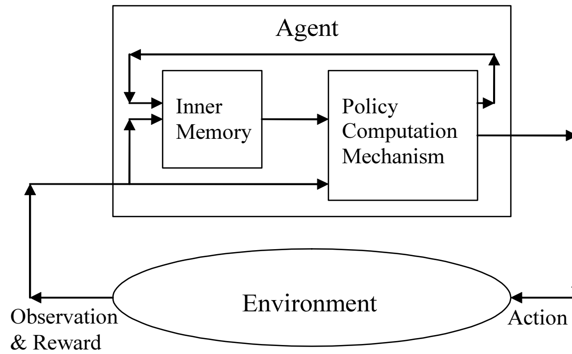

Figure 2: An agent model with an internal memory module.

For an environment with perceptually aliased states, Littman (Littman, 1994) showed that his branch-and-bound algorithm using a history window, was able to find an optimal solution, but its performance was not compared to other algorithms.

Loch et al. (Loch & Singh, 1998) also study domains where short-term memory policies are needed, and show that Sarsa(λ) using the last $2-3$ observations learns faster than Littman's branch-and-bound algorithm (Littman et al., 1995a) and achieves equal or better results than the approximation model-based algorithms in (Littman et al., 1995b) with much less computational cost. Their methods showed, though, inferior performance to stochastic policies (Williams & Singh, 1998).

7.2.2 Finite-state controllers

A Finite State Controller captures finite-size history windows smartly, by learning the optimal distinctions needed to represent the best policy graph, given its size. The FSC can be learned directly without a model of the environment. Meuleau et al.(Meuleau et al., 1999b) show FSCs to outperform reactive policies computed by the VAPS algorithm, but did not compare FSCs to any other memory-based technique.

7.2.3 Variable length histories

The use of an arbitrarily defined history of finite size does not seem like a reasonable solution to the problem of perceptual aliasing. Agents can not be expected to know how long do they need to remember their action-observation-reward trajectories. It is also usually the case that in some areas of the state space the agent needs to remember more, while in other locations a reactive policy is sufficient. A better approach would be to allow the agent to learn online the history length needed to decide on the best action in the current location.

McCallum extensively handles those issues in his dissertation (McCallum, 1996). His methods maintain the full set of past experiences (instances), clustering them together by various methods in order to learn to predict future rewards. He showed his algorithms to converge very fast on some problems with perceptual aliasing, and to produce superior results to the model-based perceptual distinctions approach of Chrisman (Chrisman, 1992), and to the neural network method Recurrent-Q (Lin & Mitchell, 1992a, 1992b). For a detailed explanation of McCallums' Instance-Based techniques see Section 7.3

7.2.4 Bayesian learning

Suematsu et al. (Suematsu, Hayashi, & Li, 1997; Suematsu & Hayashi, 1999) pointed out some problems with McCallum's methods:

- Keeping the full set of past instances can be impractical for tasks that require a long learning curve.
- Adding noisy sensors to the task definition might cause the clustering to fail to converge (though they did not demonstrate this problem).
- The learned clusters depend on the accuracy of the statistical tests, which might cause an overfit or underfit of the data.

The Bayesian learning approach requires the agent to maintain a set of candidate models, using at each step the most suitable one to compute a policy. To avoid the problems above, Suematsu et al. suggested using Bayesian learning, where the candidate models are all possible history trees — trees with the same structure as McCallum's USM, without the learning procedure. They presented an algorithm that smartly selects the current best fit history tree model, without enumerating all the possible trees explicitly. Their algorithm is incremental, and has tractable computations at each step.

Suematsu et al. provided some experimental results to demonstrate the ability of their algorithm to solve some basic POMDP environments, but compared their algorithm only to history trees with fixed depths that are not incrementally built. They provided no comparison between their algorithm and other memory based algorithms.

7.2.5 Memory bits

Another possible approach to handle perceptual aliasing is to augment the observations with some internal memory bits (Peshkin et al., 1999). The agent internal state s is therefore composed of both the current observation o and the current memory state m. The agent's action space is extended with actions that modify the current memory state by changing one of the memory bits — for every bit in memory an action is added for flipping the state of the bit on and off. The agent can apply some type of learning mechanism (such as Sarsa(λ) or VAPS) to learn the proper action, including when to change the state of the memory. The algorithm is modified so that no cost or decay is applied to the actions that modify the memory.

This approach is superior to using finite or variable history length as it can remember a meaningful event that occurred arbitrary far in the past. Keeping all the possible trajectories from the event until the outcome is observed might cost too much, and McCallum's techniques are unable to group those trajectories together to deduce the proper action. A downside to the general memory bit approach is that failing to impose any structure on the memory (such as a Finite State Controller (Meuleau et al., 1999a)) may result in a long learning curve. Peshkin et al. demonstrated their algorithms to converge, but did not show them to be superior to any other work.

Lanzi (Lanzi, 2000) also studied the effect of adding memory bits, and compared Q-learning, $Q(\lambda)$-learning — Q-learning with eligibility traces — with and without internal memory, concluding that eligibility traces and memory bits improve the performance of the agent. Lanzi did not compare his methods with other memory based techniques. Lanzi also noted that exploration of the actions that modify the memory bits should be done less seldom than exploration of actions that modify the world.

Kwee et al. (Kwee, Hutter, & Schmidhuber, 2001) used additional memory bits in market-based RL. Market-based RL assumes a population of agents that compete to produce the

highest profit (e.g. discounted sum of rewards). The agents evolve using some mechanism such as genetic algorithms. Kwee et al. demonstrated a market-based RL to solve some small POMDP domains, but did not compare it to any other approach. They also noted that this approach can not currently scale up well enough to handle larger domains.

7.2.6 Neural Networks

An appealing approach to adding an internal memory component to the agent is the use of neural networks. Lin et al. (Lin & Mitchell, 1992a) presented the Recurrent-Q algorithm, that uses a recurring neural network to learn to distinguish between perceptually aliased states.

Neural networks can be combined with evolutionary search, such as genetic algorithms, to create a population of stochastic networks that compete to generate the best possible agent. Glickman et al. (Glickman & Sycara, 2001) employed this method to several non-trivial environments and reported results superior to McCallum's U-Tree approach, though evolutionary search converges much slower than U-Tree.

Long short-term memory (LSTM) (Hochreiter & Schmidhuber, 1997) is a special recurrent neural network, with some specifically designed components called memory cells that learn to capture relevant features. LSTM was compared to Lin et al. former neural network approaches and to the memory bits approach of Peshkin et al. and converged to the optimal solution faster than both. LSTM is also the only method to the best of our knowledge that demonstrates model-free learning with noisy sensors. Most other techniques above assume either deterministic sensors (e.g. Peshkin et al., 1999), or sensors with very little noise (e.g. McCallum, 1996).

7.2.7 Hierarchical approaches

HQ-Learning (Wiering & Schmidhuber, 1997) applies to domains where the agents' task can be split into sub-tasks. In each sub-task the agent can use a reactive policy, but when transitioning from one sub-task to the other, the reactive policy changes. The perceptual aliasing problem is hence solved by placing perceptually aliased states in different tasks. HQ-Learning automatically learns the proper way to split the domain and solves the sub-tasks using Q-learning with eligibility traces. Wiering et al. showed their algorithm to outperform the standard learning algorithm with eligibility traces, but did not compare its performance to any other memory-based approaches.

Hernandez et al. (Hernandez & Mahadevan, 2000) implemented algorithms from hierarchical reinforcement learning (namely, the Hierarchy of Abstract Machines framework — HAM (Parr & Russell, 1997)) to introduce levels of abstraction to the solution. Their methods augment short-term memory techniques avoiding problems such as the similarity of history suffixes while moving back and forth in the same corridor. One of the building stones of their algorithm was the use of some memory technique (they used NSM — one of McCallum's earlier algorithms). The HSM algorithm they implemented was compared to NSM and showed superior results, but was not compared to either McCallum's USM or any other memory based techniques. It is however likely that incorporating any other basic short memory technique into the hierarchical method, HSM will be able to outperform the basic method. Hierarchical approaches seem to be very promising in real world applications, but they would still probably need to use a smart short-term memory as a subroutine.

7.3 Instance-Based Learning

As some of our algorithms we present improve on McCallum's instance-based learning algorithms, we provide some deeper background into McCallum's methods. Instance-based learning

algorithms store all past interactions of the agent with the environment in the form of tuples of action-reward-observation known as instances. Formally an instance is a tuple:

$$T_t = \langle T_{t-1}, a_t, r_t, o_t \rangle \tag{37}$$

where T_{t-1} is the previous instance and $T_0 = \perp$.

In the following sections we note some minor improvements we added to McCallum's algorithms which were used both in our base implementation and in the various augmentations to the algorithms.

7.3.1 Nearest Sequence Memory

The first instance-based algorithm McCallum suggested is Nearest Sequence Memory (NSM). In NSM the agent stores Q-values for all instances, i.e. for each instance T_t and action a we store $Q(T_t, a)$. We use some similarity metric between instances to compute the k nearest neighbors of an instance. McCallum suggest using backward identity of instances:

$$sim(T_i, T_j) = \begin{cases} 1 + sim(T_{i-1}, T_{j-1}) & , \quad a_i = a_j, o_i = o_j, r_i = r_j \\ 0 & , \quad otherwise \end{cases} \tag{38}$$

Nikovski (Nikovski, 2002) suggested the use of other similarity metrics in an offline process but these techniques are inapplicable for the online applications we focus upon.

Algorithm 6 NSM(k)

```
Input: k - the number of nearest neighbors
Initialize: t = 0, T_0 = ⊥
loop
  Compute τ - the list of k instances that are the nearest neighbors of T_t
  for all T_i ∈ τ do
    for all a ∈ A do
      Q(T_t, a) ← Q(T_t, a) + Q(T_i, a)/k
    end for
  end for
  Choose action a using an ε-greedy policy on the Q-values of the current instance T_{t-1}
  Execute a, observe reward r and observation o
  for all T_i ∈ τ do
    for all a ∈ A do
      Update: Q(T_i, a) ← (1 − α)Q(T_i, a) + α(r + γ max_{a'∈A} Q(T_{i+1}, a'))
    end for
  end for
  t ← t + 1
  Create new instance T_t = ⟨T_{t-1}, a, r, o⟩
end loop
```

Once the list of k neighbors is computed each neighbor updates the Q-values of the current instance using its own Q-values. The agent then selects the action with maximal Q-value, using some exploration technique such as ϵ-greedy and executes it, observing the next reward and observation. The agent then updates the Q-values for the k neighbors using the standard Q update rule (Equation 3).

McCallum demonstrate NSM to converge to an optimal policy on some examples where model-based techniques based on the Baum-Welch algorithm (see Section 8 for details) took an

order of magnitude more time to converge, or failed to converge at all. He also show that NSM learns faster than the Recurrent-Q neural network approach (Lin & Mitchell, 1992b). Estelle (Estelle, 2003) demonstrate NSM to learn faster than Q-learning with a fixed memory window as the problem size increases. Alexandrov (Alexandrov, 2003) show NSM to properly learn to predict[2] the next perception in a real robot navigation task outperforming bi-grams predictive model.

NSM learns a good policy very fast. Noisy observations in identical real-world experiences, though, cause the recursive similarity to terminate prematurely. In the presence of noisy sensors, therefore, the performance of NSM diminishes. NSM also cannot cleverly identify regions of the state space where it needs to look farther into the past to find important distinctions and other areas where a simple reactive policy is sufficient, and therefore introduces unnecessary similarity calculations in such simple regions.

7.3.2 Utile Suffix Memory

Utile Suffix Memory creates a tree structure, based on the well known suffix trees for string operations. This tree maintains the raw experiences and identifies matching suffixes. The root of the tree is an unlabelled node, holding all available instances. Each immediate child of the root is labelled 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 observation in the node's label. At the next level, instances are split based on the last action of the instance a_t, then on (the next to last) observation o_{t-1} and so forth. All nodes act as buckets, grouping together instances that have matching history suffixes of a certain length. Leaves take the role of 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 precept, and follow the path to the child node labelled by that precept. We then look at the action prior to this precept and move to the node labelled by that action, and then branch on the precept prior to that action and so forth, until a leaf is reached. For example, the maze in Figure 1(a) might generate the tree in Figure 7.3.2.

Identifying the proper depth for a certain leaf is a major issue. Leaves should be split if their descendants show a statistical difference in expected future discounted reward associated with the same action. Instances in a node should be split 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. Alternatively, Dutech (Dutech, 2000) suggest that leaves should be split if the Q-values do not converge. He claims that Q-value of leaves that correspond to perceptually aliased states do not converge and uses this insight to identify the leaves that should be split. Dutech however did not compare his algorithms to McCallum's basic USM.

For better performance, McCallum did not compare the nodes in the fringes to their siblings, only to their ancestor "official" leaf. He also did not compare values from all actions executed from the fringe, only the action that has the highest Q-value in the leaf (the policy action of that leaf). To compare the populations of expected discounted future rewards from the two nodes (the fringe and the "official" leaf), he used the Kolmogorov-Smirnov (KS) statistical test — a non-parametric test used to find whether two populations were generated by the same distribution. If the test reported that a statistical difference was found between the expected discounted future reward after executing the policy action, the leaf was split, the fringe node would become the new leaf, and the tree would be expanded to create deeper fringes.

[2] Alexandrov did not use NSM for control, only for prediction, as he did not keep track of Q-values.

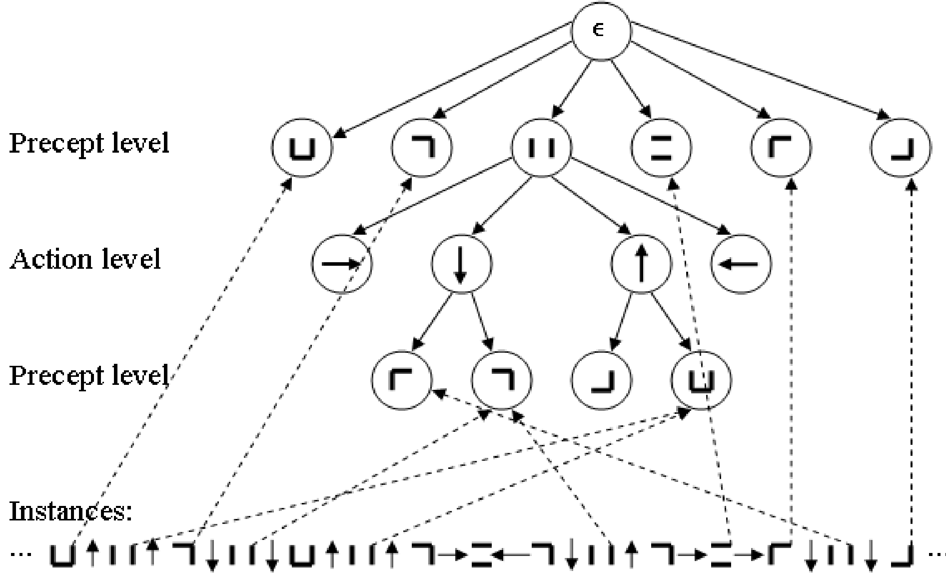

Figure 3: A possible USM suffix tree generated by the maze in Figure 1(a). Below is a sequence of instances demonstrating how some of the instances are clustered into the tree leaves.

Instead of comparing the fringe node to its ancestor "official" leaf, we (Shani & Brafman, 2004) found it computationally possible to compare the siblings of the fringe, avoiding the problem that the same instance appears in both distributions. McCallum compared only the expected discounted future rewards from executing the policy action, where we compare all the values following all actions executed after any of the instances in the fringe. Au et al. (Au & Maire, 2004) suggest to replace the KS test with an Information Gain Ratio (IGR) test, and show superior performance to the original algorithm. We also found the KS test to be insufficient and replaced it with the more robust randomization test (Yeh, 2000) that works well with small sets of instances. McCallum also considered only fringe nodes of certain depth, given as a parameter to the algorithm, where we choose to create fringe nodes as deep as possible, until the number of instances in the node diminished below some threshold (we used a value of 10 in our experiments).

The expected future discounted reward of instance T_i is defined by:

$$Q(T_i) = r_i + \gamma U(L(T_{i+1})) \tag{39}$$

where $L(T_i)$ is the leaf associated with instance T_i and $U(s) = max_a(Q(s,a))$.

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

$$R(s,a) = \frac{\sum_{T_i \in T(s,a)} r_{i+1}}{|T(s,a)|} \tag{40}$$

$$Pr(s'|s,a) = \frac{|\forall T_i \in T(s,a), L(T_{i+1}) = s'|}{|T(s,a)|} \tag{41}$$

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} Pr(s'|s,a)U(s') \tag{42}$$

This corresponds to a single step of the value iteration algorithm used in MDPs.

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:

$$a_{t+1} = argmax_a Q(L(T_t), a) \tag{43}$$

McCallum used an ϵ-greedy exploration strategy — with a probability of $1 - \epsilon$ the best action is executed and with a probability of ϵ a random action is selected.

8 Model-based Methods for POMDPs

When the model of the environment is unknown, it is possible to learn the model and then solve it using one of the solution techniques in Section 6.1. Model-based methods use past experience in the form of a sequence of instances — $\langle a_t, o_t, r_t \rangle$ — to learn a POMDP model that is likely to generate the sequence. Many researchers unfavor this approach since solving POMDPs has always been a bottleneck that needed addressing, and adding to that the difficulties of learning the model seemed to make the problem infeasible. Modern techniques however allow for a tractable solution for POMDPs with reasonably sized state spaces, making future research in the area of model-based methods more attractive.

8.1 The Baum-Welch Algorithm

Hidden Markov models are used to model dynamic systems where the transition to the next state may depend on a hidden (unobserved) variable. A hidden Markov model can be viewed as a POMDP without actions. It has been noted (Chrisman, 1992; McCallum, 1993; Nikovski & Nourbakhsh, 2000) that the Baum-Welch algorithm (e.g. Lin & Mitchell, 1997) for learning hidden Markov models can be applied to POMDPs. The agent can fix the structure of the POMDP (the number of states) and learn the state transition probabilities given the observations. The traditional approach then uses the EM algorithm to recalculate a new state space that maximizes the log-likelihood of the observations and then the Baum-Welch algorithm is executed again (e.g. Aberdeen & Baxter, 2002; Shatkay, 1999). Theocharous et al. (Theocharous, Rohanimanesh, & Mahadevan, 2001) suggested to augment this approach using hierarchical POMDPs, by modifying the Baum-Welch algorithm. Their approach seems promising, but was only tested on domains where the initial state space is initially known.

Chrisman implemented a component containing a POMDP model that predicts the current state, and updated it online using the Baum-Welch algorithm. A variation of Q-learning was used to compute Q-values given the states predicted by the POMDP predictive model, rather than solving the model. States of the POMDP could be split if the need for finer distinctions was detected. Chrisman experimented with noisy sensors and actions with stochastic results but did not compare his algorithm to any other learning method.

Maintaining a model by splitting the states that required refining in order to perform better was further explored by McCallum (McCallum, 1996, 1993) in his Utile Distinctions Memory (UDM) algorithm. McCallum initialized a model that has one state for each observation. States were then split using a statistical test on the future discounted reward of instances, given the prior state. A state is split if the statistical test showed that knowing where (which state) the agent came from helped predict future reward. The transition probabilities and policy were calculated in the same manner suggested by Chrisman. Like Chrisman, McCallum showed his algorithm to converge on a domain with some noise, but did not compare it to any other algorithm.

Wierstra et al. (Wierstra & Wiering, 2004) suggest augmenting McCallum's UDM by using an improved policy calculation algorithm that uses belief space to update Q-values and then applies a form of the Q_{MDP} approximation technique to calculate the next action. They split states when a statistically significant difference in discounted future rewards is detected using the EM algorithm to split instances into mixture components. The algorithm execution is divided into an offline learning stage, when state splitting is executed and model parameters

are re-estimated, and an online acting stage, where the belief state is maintained and Q-values updated. Their algorithm showed inferior performance to UDM on a small maze world, but was able to solve a medium sized maze, with performance similar to the RL-LSTM recurrent network algorithm with sensor accuracy of 0.7.

All algorithms that use the Baum-Welch algorithm cannot escape its heavy time complexity $O(N^2)$ for each update iteration. Using Baum-Welch to estimate the model parameters therefore seems an impractical choice for large domains. A second problem with EM and Baum-Welch is their inability to avoid local minima.

8.2 State merging methods

The algorithms we have reviewed in the previous section were initialized with a small number of states that were split if the need arose. Nikovski et al. (Nikovski & Nourbakhsh, 2000; Nikovski, 2002) suggested to start with a state for each observed instance, and merge them until a sufficient number of states has been reached. This intended state-space size is either known prior to the learning process or the algorithm can be terminated once some criteria, such as the log-likelihood of the instances given the model, drops below a predefined threshold. An initial state, corresponding to instance T_i has a probability of 1 of moving to the state representing instance T_{i+1} with a single action, single reward and single observation, but as states are merged the transition, reward and observation functions are enriched.

The first criteria Nikovski examined for merging states was the minimal decrease in log-likelihood. The initial model maximizes the log-likelihood, and every merge causes some decrease in this measure. Nikovski tried greedy merging of pairs of states that caused the minimal decrease.

Nikovski then applied one of McCallum's earlier model-free methods for discovering similar history suffixes — the Nearest Sequence Memory (NSM) algorithm — to cluster together instances. NSM identifies for every instance k other instances that agree the most on the suffix of their history and uses their Q-values to update the current instance Q-values. Nikovski clustered together the states corresponding to these instances, using some enhancements such as measuring also how much the instances agree in their future, not just their history.

Nikovski implemented these approaches for learning POMDPs, and compared them to implementations of the Baum-Welch and Steepest Gradient Ascent in log-likelihood (Binder, Koller, Russell, & Kanazawa, 1997) — algorithms where the number of states is predefined and the agent needs to learn the transition, reward and observation function. The resulting POMDPs were solved using approximation techniques such as Q_{MDP}. Nikovski did not report any attempt to apply any technique for fully solving the POMDP to the learned models. Experiments were conducted on a robot simulator with very little noise, but with perceptual aliasing. The results of the experiment show the enhanced NSM based merging technique to produce slightly superior results to the Baum-Welch and Steepest Gradient Ascent methods. Nikovski did not compare his methods to any model-free algorithm.

9 Predictive State Representations

Littman et al. (Singh, Littman, & Sutton, 2001; Singh, James, & Rudary, 2004) recently presented an alternative for POMDPs in the form of Predictive State Representations (PSRs), shown to be superior to POMDPs in terms of the ability to efficiently model problems and provide solutions. PSRs are designed to predict all possible future tests — sequences of instances — using some projection function over the given probability of observing a set of core tests. A prediction of a test is the probability of obtaining the observations in the test when executing

the test's actions. Predicting the probability of all possible future tests makes the selection of the next action easy — we can simply select the action that stochastically maximizes the sum of discounted rewards (or any other criteria) over all possible future tests. The learning algorithm for PSRs can be viewed as a two staged process:

- Discovering a good set of core tests (Rosencrantz, Gordon, & Thrun, 2004; James & Singh, 2004).
- Learn to predict the core tests (Singh, Littman, Jong, Pardoe, & Stone, 2003) and hence all possible tests.

Though PSRs were shown to provide models smaller than POMDPs that are at least as expressive, there has been little work on experimentally comparing PSRs with other model-based or with model-free algorithms. Rosencrantz et al. compare a Baum-Welch learned Hidden Markov Model with PSRs in terms of the ability to predict, demonstrating that given the model size, PSRs better predict future tests. They however did not show that a system that learns PSRs using their methods converges to an optimal policy.

References

Aberdeen, D. (2003). A (revised) survey of approximate methods for solving partially observable markov decision processes. Tech. rep., National ICT Australia, Canberra, Austalia.

Aberdeen, D., & Baxter, J. (2002). Scaling internal-state policy-gradient methods for pomdps. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pp. 1–12. Morgan Kaufmann.

Alexandrov, S. (2003). Ratbert: Nearest sequence memory based prediction model applied to robot navigation. In *Proceedings of the 20th International Conference on Machine Learning*.

Au, M., & Maire, F. (2004). Automatic state construction using decision tree for reinforcement learning agents. In *Proceedings of the International Conference on Computational Intelligence for Modelling Control and Automation*.

Baird, L., & Moore, A. (1998). Gradient descent for general reinforcement learning. In *Proceedings of the 11th International Conference on Neural Information Processing Systems*, pp. 968–974. MIT Press.

Baird, L. C. (1999). *Reinforcement Learning Through Gradient Descent*. Ph.D. thesis, Carnegie Mellon University.

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Inteligence*, *72(1)*, 81–138.

Bellman, R. E. (1962). *Dynamic Programming*. Princeton University Press.

Binder, J., Koller, D., Russell, S. J., & Kanazawa, K. (1997). Adaptive probabilistic networks with hidden variables. *Machine Learning*, *29*(2-3), 213–244.

Bonet, B., & Geffner, H. (2003). Labeled rtdp: Improving the convergence of real-time dynamic programming. In *Proceedings of the 13Th International Conference on Automated Planning and Scheduling*. AAAI Press.

Braziunas, D., & Boutilier, C. (2003). Stochastic local search for pomdp controllers. In *Proceedings of The Nineteenth National Conference on Artificial Intelligence (AAAI-04)*. Morgan Kaufman.

Cassandra, A., Littman, M. L., & Zhang, N. L. (1997). Incremental pruning: A simple, fast, exact algorithm for partially observable markov decision processes. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence*, pp. 54–61. Morgan Kaufmann.

Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *National Conference on Artificial Intelligence*, pp. 183–188.

Dutech, A. (2000). Solving pomdps using selected past events. In *Proceedings of the Fourteenth European Conference on Artificial Intelligence*, pp. 281–285. IOS Press.

Estelle, J. (2003). Reinforcment learning in pomdps: Instance-based state identification vs. fixed memory representations. Tech. rep., Cornell University.

Glickman, M. R., & Sycara, K. P. (2001). Evolutionary search, stochastic policies with memory, and reinforcement learning with hidden state. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pp. 194–201. Morgan Kaufmann.

Hansen, E. A. (1998). Solving pomdps by searching in policy space. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pp. 211–219. Morgan Kaufmann.

Hasinoff, S. W. (2002). Reinforcement learning for problems with hidden state. Tech. rep., University of Toronto, Department of Computer Science.

Hayashi, A., & Suematsu, N. (1999). Viewing Classifier Systems as Model Free Learning in POMDPs. In *Advances in Neural Information Processing Systems*, pp. 989–995.

Hernandez, N., & Mahadevan, S. (2000). Hierarchical memory-based reinforcement learning. In *Proceedings of the Fifteenth International Conference on Neural Information Processing Systems*, pp. 1047–1053. Morgan Kaufmann.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780.

Howard, R. A. (1960). *Dynamic Programming and Markov Processes.* MIT Press.

Jaakkola, T., Singh, S. P., & Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable Markov decision problems. In Tesauro, G., Touretzky, D., & Leen, T. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 7, pp. 345–352. MIT Press.

James, M., & Singh, S. (2004). Learning and discovery of predictive state representations in dynamical systems with reset. In *Proceedings of the 21st International Conference on Machine Learning.* ACM Press.

Kaelbling, L. P., Littman, M., & Moore, A. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Inteligence*, *101*, 99–134.

Kwee, I., Hutter, M., & Schmidhuber, J. (2001). Market-based reinforcement learning in partially observable worlds. In *Proceedings of the International Conference on Artificial Neural Networks*, pp. 865–873. Springer-Verlag.

Lanzi, P. L. (2000). Adaptive Agents with Reinforcement Learning and Internal Memory. In *Proceedings of the Sixth International Conference on the Simulation of Adaptive Behavior*, pp. 312–322. MIT Press.

Lin, L.-J., & Mitchell, T. M. (1992a). Memory approaches to reinforcement learning in non-markovian domains. Tech. rep. CMU-CS-92-138, Carnegie Mellon University, Pittsburgh, PA 15213.

Lin, L. J., & Mitchell, T. M. (1992b). Reinforcement learning with hidden states. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pp. 271–280. MIT Press.

Lin, L. J., & Mitchell, T. M. (1997). A gentle tutorial on the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models. Tech. rep. ICSI-TR-97-021, Barkley, CA.

Littman, M. L. (1994). Memoryless policies: Theoretical limitations and practical results. In Cliff, D., Husbands, P., Meyer, J.-A., & Wilson, S. W. (Eds.), *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, pp. 238–245 Cambridge, MA. MIT Press.

Littman, M. L., Cassandra, A. R., & Kaelbling, L. P. (1995a). Efficient dynamic-programming updates in partially observable markov decision processes. Tech. rep. CS-95-19, Brown University.

Littman, M. L., Cassandra, A. R., & Kaelbling, L. P. (1995b). Learning policies for partially observable environments: Scaling up. In Prieditis, A., & Russell, S. (Eds.), *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 362–370 San Francisco, CA, USA. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

Loch, J., & Singh, S. (1998). Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes. In *Proceedings 15th International Conference on Machine Learning*, pp. 323–331. Morgan Kaufmann, San Francisco, CA.

McCallum, A. K. (1993). Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 190–196 Amherst, MA.

McCallum, A. K. (1996). *Reinforcement Learning with Selective Perception and Hidden State*. Ph.D. thesis, Department of Computer Science, University of Rochester.

Meuleau, N., Kim, K. E., Kaelbling, L. P., & Cassandra, A. R. (1999a). Solving pomdps by searching the space of finite policies. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pp. 417–426. Morgan Kaufmann.

Meuleau, N., Peshkin, L., Kim, K., & Kaelbling, L. P. (1999b). Learning finite-state controllers for partially observable environments. In *Proceedings of the Fifteenth Conference on Uncertainity in Artificial Intelligence*, pp. 427–436. Morgan Kaufmann.

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

Murphy, K. (2000). A survey of pomdp solution techniques. Tech. rep., U.C. Berkely.

Nikovski, D. (2002). *State-Aggregation Algorithms for Learning Probabilistic Models for Robot Control*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

Nikovski, D., & Nourbakhsh, I. (2000). Learning probabilistic models for decision-theoretic navigation of mobile robots. In *Proceedings of the 17th International Conference on Machine Learning*, pp. 671–678. Morgan Kaufmann, San Francisco, CA.

Parr, R., & Russell, S. (1995). Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1088–1094.

Parr, R., & Russell, S. (1997). Reinforcement learning with hierarchies of machines. In *In Proceedings of Advances in Neural Information Processing Systems*, Vol. 10, pp. 1043–1049. MIT Press.

Peshkin, L., Meuleau, N., & Kaelbling, L. P. (1999). Learning policies with external memory. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pp. 307–314 San Francisco, CA, USA. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

Poupart, P. (2002). *Partially Observable Markov Decision Processes*. Ph.D. thesis, University of Toronto.

Poupart, P., & Boutilier, C. (2004). Bounded finite state controllers. In *Proceedings of the 16th International Conference on Neural Information Processing Systems*. MIT Press.

Poupart, P., Boutilier, C., Schuurmans, D., & Patrascu, R. (2002). Piecewise linear value function approximation for factored mdps. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI02)* Edmonton.

Puterman, M. (1994). *Markov Decision Processes*. Wiley, New York.

R. S. Sutton, A. G. B. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

Rosencrantz, M., Gordon, G., & Thrun, S. (2004). Learning low dimensional predictive representations. In *Proceedings of the Twenty-First International Conference on Machine Learning* Banff, Alberta, Canada.

Shani, G., & Brafman, R. I. (2004). Resolving perceptual aliasing in the presence of noisy sensors. In *Proceedings of the Eighteenth Annual Conference on Neural Information Processing Systems*. MIT Press.

Shatkay, H. (1999). Learning hidden markov models with geometrical constraints. In *In Proceedings of the International Conference on Uncertainty in Artificial Intelligence (UAI)*, pp. 602–611.

Singh, S., James, M. R., & Rudary, M. R. (2004). Predictive state representations: A new theory for modeling dynamical systems. In *Proceedings of the Twentieth International Conference on Uncertainty in Artificial Intelligence*, pp. 512–519.

Singh, S., Littman, M. L., Jong, N. K., Pardoe, D., & Stone, P. (2003). Learning predictive state representations. In *Proceedings of the Twentieth International Conference on Machine Learning*, pp. 712–719.

Singh, S., Littman, M. L., & Sutton, R. S. (2001). Predictive representations of state. In *Proceedings of the 15th International Conference on Neural Information Processing Systems*, pp. 1555–1561. MIT Press.

Singh, S. P., & Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, *22*(1–3), 123–158.

Sondik, E. J. (1971). *The Optimal Control of Partially Observable Markov Processes*. Ph.D. thesis, Stanford University.

Suematsu, N., & Hayashi, A. (1999). A reinforcement learning algorithm in partially observable environments using short-term memory. *Advances in Neural Information Processing Systems*, *11*, 1059–1065.

Suematsu, N., Hayashi, A., & Li, S. (1997). A Bayesian approach to model learning in non-Markovian environments. In *Proceedings of the 14th International Conference on Machine Learning*, pp. 349–357. Morgan Kaufmann.

Sutton, R. S. (1991). Planning by incremental dynamic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 353–357. Morgan Kaufmann.

Theocharous, G., Rohanimanesh, K., & Mahadevan, S. (2001). Learning hierarchical partially observable markov decision processes for robot navigation. In *Proceedings of the IEEE Conference on Robotics and Automation (ICRA)*.

Watkins, C., & Dayan, P. (1992). Q-learning. *Journal of Machine Learning*, *8*, 279–292.

Whitehead, S. D., & Ballard, D. H. (1991). Learning to perceive and act by trial and error. *Journal of Machine Learning*, *7*, 45–83.

Wiering, M., & Schmidhuber, J. (1997). Hq-learning. *Adaptive Behavior*, *6*(2), 219–246.

Wierstra, D., & Wiering, M. (2004). Utile distinction hidden markov models. In *Proceedings of the Twenty-First International Conference on Machine Learning*.

Williams, J. K., & Singh, S. (1998). Experimental results on learning stochastic memoryless policies for partially observable markov decision processes. In *Proceedings 11th Conference on Neural Information Processing Systems*, pp. 1073–1079. The MIT Press.

Yeh, A. (2000). More accurate tests for the statistical significance of result differences. In *Proceedings of the Eighteenth International Conference on Computational Linguistics*, pp. 947–953.