Instance-Based Utile Distinctions for Reinforcement Learning with Hidden State

R. Andrew McCallum
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
`mccallum@cs.rochester.edu`

Abstract

We present *Utile Suffix Memory*, a reinforcement learning algorithm that uses short-term memory to overcome the state aliasing that results from hidden state. By combining the advantages of previous work in instance-based (or "memory-based") learning and previous work with statistical tests for separating noise from task structure, the method learns quickly, creates only as much memory as needed for the task at hand, and handles noise well.

Utile Suffix Memory uses a tree-structured representation, and is related to work on Prediction Suffix Trees [Ron *et al.*, 1994], Parti-game [Moore, 1993], G-algorithm [Chapman and Kaelbling, 1991], and Variable Resolution Dynamic Programming [Moore, 1991].

1 INTRODUCTION

The sensory systems of embedded agents are inherently limited. When a reinforcement learning agent's sensory limitations hide features of the environment from the agent, we say that the agent suffers from *hidden state*.

There are many reasons why important features can be hidden from a robot's perception: sensors have noise, limited range and limited field of view; occlusions hide areas from sensing; limited funds and space prevent equipping the robot with all desired sensors; an exhaustible power supply deters the robot from using all sensors all the time; and the robot has limited computational resources for turning raw sensor data into usable percepts.

The hidden state problem arises as a case of *perceptual aliasing*: the mapping between states of the world and sensations of the agent is not one-to-one [Whitehead and Ballard, 1991]. If perceptual limitations allow the agent to perceive only a portion of its world, then many different world states can produce in the same percept. Also, if the agent has an active perceptual system, meaning that it can redirect its sensors to different parts of its surroundings, then the reverse will also be true—many different percepts can result from the same world state.

Perceptual aliasing is both a blessing and a curse. It is a blessing because it can provide useful invariants by representing as equivalent world states in which the same action is required. It is a curse because it can also confound world states in which *different* actions are required. Perceptual aliasing provides powerful generalization, but it can also over-generalize. The trick is to *selectively* remove hidden state, so as to uncover the hidden state that impedes task performance, but leave ambiguous the hidden state that that is irrelevant to the agent's task. Distinguishing states whose difference is irrelevant to the current task not only causes the agent to uselessly increase its storage requirements, more damagingly, it also prolongs learning time by requiring that the agent re-learn its policy in each of the needlessly distinguished states.

State identification techniques use history information to uncover hidden state [Bertsekas and Shreve, 1978]. Instead of defining agent internal state by percepts alone, the agent defines its internal state space by a combination of a percepts and short-term memory of past percepts and actions. If the agent uses enough short-term memory in the right places, the agent can uncover the non-Markovian dependencies that caused the task-impeding hidden state.

Predefined, fixed-sized memory representations are often undesirable. When memory size (*i.e.* number of internal state variables) is more than needed, it exponentially increases the number of agent internal states for which a policy must be learned and stored; when the size of the memory is less than needed, the agent reverts to the disadvantages of undistinguished hidden state. Even if the agent designer understands the task well enough to know its maximal memory requirements, the agent is at a disadvantage with fixed-sized memory because, for most tasks, different amounts of memory are needed at different steps of the task. We conclude that the agent should learn on-line how much memory is needed for different parts of its state space.

The work described in this paper addresses the issue of hidden state in conjunction with the following principles:

- Agents should learn their tasks. We would prefer that our agents learn their behaviors rather than have them hard-coded because programming all the details by hand is tedious, we may not know the environment ahead of time, and we want the robot to adapt its behavior as the environment changes.
- Agents should learn in as few trials as possible. Learning experience is expensive in terms of wall clock time, and dangerous in terms of potential damage to the robot and its surroundings. Experience is often relatively more expensive than storage and computation. Furthermore, the expense of computation and storage will only decrease as advances in computing hardware continue; the cost of experience will not.
- Agents should be able to handle noisy perceptions, actions and rewards.
- Agents should use short-term memory to uncover task-relevant hidden state. By working with an internal state representation that is non-Markovian with respect to rewards and actions, the agent is, by definition, not able to predict rewards and choose actions as well as it could if it used a Markovian state representation. A Markovian state representation is obtainable by using memory.

  Furthermore, the agent should *learn* how much memory to use for the same reasons cited in the first principle above; and, the agent should make only task-relevant distinctions because making more distinctions than necessary wastes learning time, as explained earlier.

Some Previous Work in Reinforcement Learning with Hidden State

Some solutions to the hidden state problem do not use short-term memory, but do as best they can using a non-Markovian state representation. The *Lion* algorithm, for example, simply avoids passing through aliased states. Whenever the agent finds a state that delivers inconsistent reward, it sets that state's utility so low that the policy will never visit it again. The success of this algorithm depends on a deterministic world and on the existence of a path to the goal that consists of only unaliased states. Some other solutions do not avoid aliased states, but involve either learning deterministic policies that execute incorrect actions in some aliased states, or learning stochastic policies that, in aliased states, may execute incorrect actions with some probability. [Littman, 1994; Jaakkola *et al.*, 1995].

Several reinforcement learning algorithms have adopted an approach that learns to use memory in defining the agent's internal state space. The Perceptual Distinctions Approach [Chrisman, 1992] and Utile Distinction Memory [McCallum, 1993] are both based on splitting states of a finite state machine by doing batched analysis of statistics gathered over many steps. Recurrent-Q [Lin, 1993] is based on training recurrent neural networks. Genetically Programmed Indexed Memory [Teller, 1994] evolves agents that use load and store instructions on a register bank. A chief disadvantage of all these techniques is that they require a very large number of steps for training. Of these memory-using algorithms, only two of them, the Perceptual Distinctions Approach and Utile Distinction Memory, augment their memory capacity on-line during learning.

Two Complementary Algorithms

Utile Distinction Memory has an additional interesting feature in that it only adds new memory capacity where doing so will increase the agent's ability to predict reward. Unlike schemes based on predicting percepts, UDM uses future discounted reward to build an agent internal state space that is only as large as needed to perform the task at hand—not as large as needed to represent the entire perceived world, including all the irrelevant details. If the agent's task is simple, but the world is complex, we would still like the agent's internal state space to be simple. UDM attains this task-dependent representation using a robust statistical test to distinguish between reward variations that could be predicted if new memory were added, and reward variations that are due to unpredictable noise.

UDM is not without its problems, however. In addition to learning extremely slowly, it has trouble discovering the utility of memories longer than one time step since the statistical test only examines the benefit of making a single additional state split at a time.

These concerns are addressed by a recent algorithm called *Nearest Sequence Memory* (NSM) [McCallum, 1995]. It comes from a family of techniques dubbed *Instance-Based State Identification*—techniques that use instance-based (or "memory-based") learning for uncovering hidden state. NSM, specifically, is based on k-nearest neighbor.

The key idea behind Instance-Based State Identification is the recognition that recording raw experience is particularly advantageous when learning to partition a state space, as is the case when the agent is trying to determine how much history is significant for uncovering hidden state. If, during learning, the agent incorporates experience merely by averaging it into its current, flawed state space partitions, it is bound to attribute experience to the wrong states; experience attributed to the wrong state turns to garbage and is wasted. Recorded raw experience, however, can be reinterpreted whenever state-space boundaries shift during learning. When faced with an evolving state space, keeping raw previous experience is the path of least commitment, and thus the most cautious about losing information. NSM learns about an order of magnitude faster than several other hidden state reinforcement learning algorithms [McCallum, 1995].

NSM's other advantage over UDM is that it can easily create memories that are multiple steps long. Since the algorithm records the raw experience in sequence, all the steps leading up to a reward are available for examination.

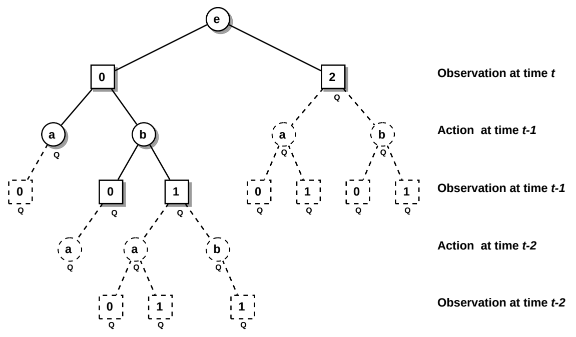


Figure 1: A USM-style Suffix Tree. Percepts are indicated by integers, actions by letters. The fringe nodes are drawn in dashed lines. Nodes labeled with a Q are nodes which hold Q-values.

Ironically, however, NSM falls severely short in exactly those areas where UDM does so well. Like k-nearest neighbor, NSM cannot explicitly separate variations due to noise from variations due to task structure; and thus, NSM does not handle noise well. Furthermore, NSM has no concept of "utile distinctions"—the amount of short-term memory NSM creates depends erroneously on regional sample density, not on the requirements of the task.

2 UTILE SUFFIX MEMORY

This paper describes a new method for dynamically adding short-term memory to the state representation of a reinforcement learning agent. We call it *Utile Suffix Memory*, (USM). It is the result of an effort to combine the advantages of instance-based learning and utile distinctions. Like Nearest Sequence Memory, the algorithm records raw experience; like Utile Distinction Memory, the algorithm determines how much memory to consider significant by using a statistical test based on future discounted reward. The technique makes efficient use of raw experience in order to learn quickly and discover multi-step memories easily, but it also uses a statistical technique in order to separate noise from task structure and build a task-dependent state space.

Like all instance-based algorithms, USM records each of its raw experiences. For a reinforcement learning agent, these are transitions consisting of action-percept-reward triples connected in a time-ordered chain. We will call these individual raw experiences *instances*. Unlike NSM, USM organizes, or "clusters," these instances in such a way as to explicitly control how much of their history to consider significant. The structure that controls this clustering is a kind of finite state machine called a Prediction Suffix Tree [Ron *et al.*, 1994]. The structure can be thought of as an order-n Markov model, with varying n in different parts of state space.

A leaf of the tree acts as a bucket, grouping together instances that have matching history up to a certain length. The deeper the leaf in the tree, the more history the instances in that leaf share. The branches off the root of the tree represent distinctions based on the current percept; the second-level branches add distinctions based on the immediately previous action; the next branches use the previous percept, and so on, backwards in time. To add a new instance to the tree, we first examine its percept, and follow the corresponding root branch; then we look at the action of the new instance's predecessor in the instance chain, and follow the corresponding second-level branch; then we examine the percept of the new instance's predecessor; we proceed likewise until we reach a leaf, and deposit the new instance there.

The agent builds this tree on-line during training—beginning with no short-term memory, and selectively adding branches only where additional memory is needed. In order to calculate statistics about the value of additional distinctions, the tree includes a "fringe," or additional branches below what we normally consider the leaves of the tree. The instances in the fringe nodes are tested for statistically significant differences in expected future discounted reward on a per-action basis. If the Kolmogorov-Smirnov test indicates that the instances came from different distributions, these fringe nodes become "official" leaves, and the fringe is extended below them. When deeper nodes are added, they are populated with instances from their parent node, the instances being properly distributed among the new children according to the additional distinction. The depth of the fringe is a configurable parameter. By using a multi-step fringe USM can successfully discover useful multi-step memories.

3 DETAILS OF THE ALGORITHM

The interaction between the agent and its environment is described by actions, observations and rewards. There is a finite set of possible actions, $\mathcal{A} = \{a^1, a^2, \ldots, a^{|\mathcal{A}|}\}$, a finite set of possible observations[1], $\mathcal{O} = \{o^1, o^2, \ldots, o^{|\mathcal{O}|}\}$, and scalar range of possible rewards, $\mathcal{R} = [x, y], x, y \in \Re$. At each time step, t, the agent executes an action, $a_t \in \mathcal{A}$, then as a result receives an observation, $o_{t+1} \in \mathcal{O}$, and a reward, $r_{t+1} \in \mathcal{R}$. We will use subscript i to indicate an arbitrary time.

The agent aims to learn an action-choosing policy that maximizes its expected discounted sum of future reward, called return, and written $\mathbf{r}_t$.

$$\mathbf{r}_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^n r_{t+n} + \ldots \quad (1)$$

Like other instance-based algorithms, Utile Suffix Memory records each of its raw experiences. The experience associated with time t is captured as a transition "instance" in four dimensional space, written T_t. The instance is a tuple consisting of all the available information associated with the transition to time t: the previous instance in the

[1]Section 6, and in more detail [McCallum, 1994], discusses a natural extension for handling multi-dimensional continuous spaces.

chain, the action, a_{t-1}, the resulting observation, o_t, and the resulting reward, r_t.

$$T_t = < T_{t-1}, a_{t-1}, o_t, r_t > \quad (2)$$

We will write T_{i-1} to indicate T_i's predecessor in the instance chain, T_{i+1} to indicate T_i's successor.

In addition to organizing the instances in a time-ordered chain, Utile Suffix Memory also clusters the instances in the nodes of a suffix tree.

Tree nodes are labeled such that deeper layers of the tree add distinctions based alternately on previous observations and actions. Nodes at odd depths are labeled with an observation, $o \in \mathcal{O}$, and nodes at even depths are labeled with an action, $a \in \mathcal{A}$; no two children of the same parent have the same label. Each node of the tree can thus be uniquely identified by the string of labels on the path from the node to the root. This string, s, is called the node's *suffix*. We will use s interchangeably to indicate the suffix as well as the tree node that the suffix specifies.

An instance, T, is deposited in the leaf node whose suffix, s, matches some suffix of the actions and observations of the transition instances that precede T in time. That is, transition T_t belongs to the leaf with a label that is some suffix of $[..., o_{t-3}, a_{t-3}, o_{t-2}, a_{t-2}, o_{t-1}]$. The set of instances associated with the leaf labeled s is written $\mathcal{T}(s)$. The suffix tree leaf to which instance T belongs is written $L(T)$.

Below the "official" leaves of the suffix tree are additional layers of nodes called a fringe. Fringe nodes are labeled by the same scheme as the non-fringe nodes; they also contain transitions according to the same suffix criterion used by non-fringe nodes. The purpose of the fringe is to provide the agent with "hypothesis" distinctions—by performing tests on the distinctions introduced by the fringe, the agent can decide whether or not to promote fringe distinctions to the status of "official distinction" used to partition its state space. The fringe may be as deep or shallow as desired; and different branches of the fringe may have different depths. A deeper fringe will help the agent discover conjunctions of longer, multi-step distinctions. We will use the term "leaf" to refer only to *non-fringe*, "official" leaves.

Theoretically, the number of fringe nodes would grow exponentially with depth of the fringe, but in practice, the growth of the fringe is not this extreme because fringe (and non-fringe) nodes need not be created until the agent actually experiences transition sequences with suffixes that match those nodes. In practice, many sequences are never experienced—either because they are impossible in the given environment, or because the reinforcement learner's on-line balance of exploitation versus exploration never visits them. For example, all possible sequences leading up to bumping into a wall would not be explored. In any case, there will always be fewer tree nodes than instances.

In USM, the leaves of the suffix tree are the internal states of the reinforcement learning agent. Thus, deep branches of the tree correspond to regions of the agent's internal state space that are "finely distinguished", with specific, long memories; and shallow branches of the tree correspond to regions of the agent's internal state space that are only "broadly distinguished", with little or no memory. The agent maintains learned estimates of expected future discounted reward for each state-action pair. The estimate, or "Q-value", for choosing action a from leaf with suffix s is written $Q(s, a)$. Note that, although deeper layers of the tree correspond to earlier time steps, all Q-values indicate the expected values for the next step in the future.

Here are the steps of the USM algorithm:

1. The agent begins with a tree that represents no history information, *i.e.*, the tree makes distinctions based only on the agent's current percept. This tree has a root node with one child for each percept, and all its child nodes are leaves. Below these leaves, the tree also has a fringe of the desired depth.

   For all tree nodes s, $\mathcal{T}(s)$ is empty. The time-ordered chain of instances is empty.

2. The agent makes a step in the environment. It records the transition as an instance, and puts the instance on the end of the chain of instances. That is, create the transition instance, T_t:

$$T_t = < T_{t-1}, a_{t-1}, o_t, r_t > \quad (3)$$

   If one is concerned about the size of the instance chain, we can limit its growth by simply discarding the oldest instance before adding each new instance, once some reasonably sized limit has been reached. Additionally this can help deal with a changing environment.

   The agent also associates the new instance with the leaf and fringe nodes that have suffixes matching a suffix of the new instance. These nodes are the set of nodes along the path from the suffix-matching fringe leaf node up to the leaf node. Thus, the transition is deposited in only as many nodes as the fringe is deep. For all s, such that s is a fringe or leaf node and s is a suffix of T_t:

$$\mathcal{T}(s) \leftarrow \mathcal{T}(s) \cup \{T_t\} \quad (4)$$

3. For each step in the world, the agent does one step of value iteration [Bellman, 1957], with the leaves of the tree as states. If computational limitations or the number of states makes full value iteration too expensive, we can instead do Prioritized Sweeping [Moore and Atkeson, 1993], Q-DYNA [Peng and Williams, 1992] or even Q-learning [Watkins, 1989]. However, the small number of agent internal states created by USM's "utile distinction" nature often makes value iteration feasible where it would not have previously been possible with typically exponential fixed-granularity state space divisions.

   Value iteration consists of performing one step of dynamic programming on the Q-values:

$$Q(s,a) \leftarrow R(s,a) + \gamma \Pr(s'|s,a)U(s') \quad (5)$$

   where $R(s,a)$ is the estimated immediate reward for executing action a from state s, $\Pr(s'|s,a)$ is the estimated probability that the agent arrives in state s' given that it executed action a from state s, and $U(s')$ is the utility of state s', calculated as $U(s') = \max_{a \in A} Q(s', a)$.

   Both $R(s,a)$ and $\Pr(s'|s,a)$ can be calculated directly from the recorded instances. Let $\mathcal{T}(s,a)$ be the set of all transition instances in the node s that also record having executed action a. Remember that r_i is recorded as an element of T_i.

$$R(s,a) = \frac{\sum_{T_i \in \mathcal{T}(s,a)} r_i}{|\mathcal{T}(s,a)|} \quad (6)$$

$$\Pr(s'|s,a) = \frac{|\forall T_i \in \mathcal{T}(s,a) \ s.t. \ L(T_{i+1}) = s'|}{|\mathcal{T}(s,a)|} \quad (7)$$

The efficiency of the value-iteration calculation can obviously benefit from some simple caching of values for $R(s,a)$ and $\Pr(s'|s,a)$, using a strategy that incrementally updates these values whenever a new instance is added to the relevant tree node.

Note that, although USM explicitly represents reward noise and action noise through its calculation of $R(s,a)$ and $\Pr(s'|s,a)$, it does not explicitly represent perception noise. This does not imply that USM cannot handle noisy percepts—it can handle perception noise as well as other techniques, such as Q-learning, that simply average together outcomes from matching percepts. USM would, however, handle perception noise better if it explicitly represented it, as do partially observable Markov decision processes (POMDP's), for example. There has been some work with POMDP's that find an optimal policy *given* an observation, transition and reward model of the environment, for example: [White and Scherer, 1994] uses a fixed-length suffix of the percept-action sequence; [Platzman, 1977] uses a variable-length finite suffix; [Cassandra *et al.*, 1994] learns a policy without a finite suffix. Investigating the combination of methods that learn a model on line, such as USM, and methods that learn policies for POMDP's is the subject of future work.

4. After adding the new instance to the tree the agent determines whether this new information warrants adding any new history distinctions to the agent's internal state space. The agent examines the fringe nodes under the new instance's leaf node, and uses the Kolmogorov-Smirnov test to compare the distributions of future discounted reward associated with the same action from different nodes. If the test indicates that two distributions have a statistically significant difference, this implies that promoting the relevant fringe nodes into non-fringe leaf nodes will help the agent predict reward. Thus the agent only introduces new history distinctions when doing so will help the agent predict reward.

   The Kolmogorov-Smirnov test answers the question, "are two distributions significantly different," as does the Chi-Square test, except that the Kolmogorov-Smirnov test works on unbinned data, a feature that is necessary since the agent is comparing distributions of real-valued numbers—expected future discounted rewards.

   The distribution of expected future discounted reward associated with a particular node is composed of the set of expected future discounted reward values associated with the individual instances in that node. The expected future discounted reward of instance T_i is written $Q(T_i)$, and is defined as:

$$Q(T_i) = r_i + \gamma U(L(T_{i+1})) \quad (8)$$

   Instead of comparing the distributions of the fringe nodes against each other, (a process that would result in n^2 comparisons, where n is the number of fringe nodes under the new instance's leaf), the current implementation instead compares the distributions of the fringe nodes against the distribution in the leaf node (resulting in only n comparisons).

   If the tests result in promoting fringe nodes into leaves, the fringe is extended below the new leaves as necessary to preserve the desired fringe depth. Note that, since this is an instance-based algorithm, when new distinctions are added, previous experience can be properly partitioned among the new distinctions: when new nodes are added to the tree, the instances in the parent are correctly distributed among the new children by looking one extra time step further back into each instance's history.

   If concerned about computation time, the agent could perform the test only once every n instance additions to a leaf node, instead of testing after every addition.

5. The agent chooses its next action based on the Q-values in the leaf corresponding to its recent history of actions and observations. That is, it chooses a_{t+1} such that

$$a_{t+1} = \text{argmax}_{a \in A} Q(L(T_t), a) \quad (9)$$

   Alternatively, with probability e, the agent explores by choosing a random action instead.

   Increment t and return to step 2.

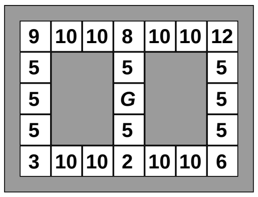
4 EXPERIMENTAL RESULTS

Utile Suffix Memory has been tested in several environments; here we show results from three of them: (1) a hallway navigation task in which over 70% of the states are perceptually aliased and the agent also suffers from noisy rewards, actions and percepts, (2) the space docking task used previously to test the performance of the Perceptual Distinctions Approach [Chrisman, 1992], as well as Nearest Sequence Memory [McCallum, 1995], and (3) a "Blocks World" hand-eye coordination task, very much like that used in [Whitehead and Ballard, 1991], but with more hidden state.

4.1 HALLWAY NAVIGATION

The agent's task is to navigate to the goal location using actions for moving north, south, east and west. A key difficulty is that the robot's sensors provide only local information: binary indications the presence or absence of a barrier immediately adjacent to the robot in each of the four compass directions. Figure 2 (top) shows a map of the environment; note that different actions are required in many of the perceptually aliased states. The agent receives reward 5.0 upon reaching the goal, reward -1.0 for attempting to move into a barrier, and reward -0.1 otherwise. After reaching the goal, the agent executes any action and begins a new trial in a randomly chosen corner of the world.

A graph of performance during learning is shown in figure 2 (bottom). The agent used a temporal discount factor, $\gamma = 0.9$, a constant exploration probability $e = 0.1$, and a fringe of depth 3. USM learns quickly and handles noise well. Action noise, when added, consisted of executing a random action with probability 0.1, instead of the action



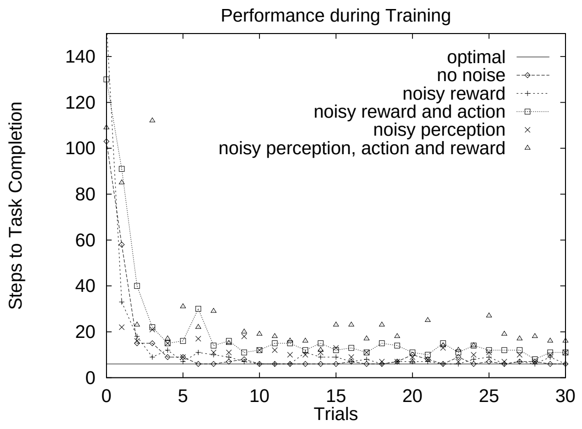


Figure 2: **Top:** A hallway navigation task with limited sensors. World locations are labeled with integers that encode the four bits of perception. **Bottom:** Performance during learning, with and without noise. The plot shows the median of ten runs.

chosen by the agent's policy; perceptual noise consisted of seeing a random observation with probability 0.1, and reward noise consisted of adding values uniformly chosen from the range -0.1 to $+0.1$ to the normal reward. For all combinations of noise, the agent learned the policy that would be optimal if the action and perception noise did not interfere. In the graph, some suboptimal trial lengths continue to occur late in learning due to exploration steps, action noise and perception noise.

Many of the smarter exploration techniques, such as various counter-based techniques [Thrun, 1992], which would have caused exploration to decay after an appropriate amount of experience, do not apply straightforwardly to tasks with hidden state. Efficient exploration with hidden state is an extremely difficult problem, and remains an area requiring work; exploration issues are not addressed at all in this paper. Some of the problems regarding exploration with hidden state are discussed further in [McCallum, 1994].

Figure 3 shows a suffix tree learned by Utile Suffix Memory. USM successfully separates the noise in the environment from the non-Markov structure in the environment, thus building only as much memory as needed to perform the task at hand, not as much as needed to model the entire environment, and definitely not as much as needed to represent uniformly-deep memories everywhere. Deep branches are created where needed to provide the multi-step memory required for disambiguating the various state 10's; shallower branches are created for the one-step memory needed to

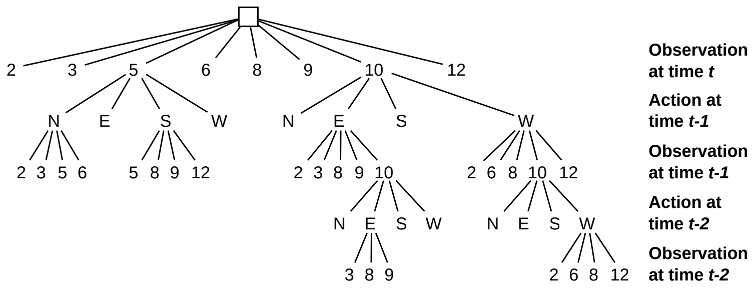


Figure 3: A tree learned by Utile Suffix Memory for navigating through the environment in figure 2. Deeper branches of the tree correspond to longer, more detailed memories. USM created memory only where needed to solve the task at hand.

disambiguate the two state 5's on the critical path for this task, (but the deeper branches that would be needed to disambiguate the state 5's on the sides of the environment are not created because they are not needed to solve the task); and no memory is created at all for the states that are not aliased.

4.2 PERFORMANCE IN CHRISMAN'S SPACESHIP DOCKING TASK

The Perceptual Distinctions Approach [Chrisman, 1992] was demonstrated in a spaceship docking application with hidden state. The task is made difficult by noisy sensors and unreliable actions. Some of the sensors returned incorrect values 30 percent of the time. Various actions failed 70, 30 or 20 percent of the time, and when they failed, resulted in random states. See [Chrisman, 1992] for a full description. Nearest Sequence Memory [McCallum, 1995] was also tested in the same environment.

A graph comparing performance during learning for each of these three algorithms is shown in figure 4. USM used a temporal discount factor, $\gamma = 0.9$, a constant exploration probability $e = 0.1$, and a fringe of depth 3. The sum of discounted reward is plotted on the vertical axis against the number of steps taken since the beginning of learning on the horizontal axis. The Perceptual Distinctions Approach takes about 8000 steps to learn the task; in order to maintain resolution in the graph, only the first 1000 steps are shown. Nearest Sequence Memory learns the task in an order of magnitude less time, about 600 steps. Utile Suffix Memory learns the task in approximately half the time taken by Nearest Sequence Memory, about 300 steps. Some of the reasons for this better performance are discussed briefly in section 1, and more fully in [McCallum, 1994].

Utile Suffix memory also does not require more computation time. It is difficult to compare the time complexity of the algorithms directly, since they are based on different underlying constants, but the following analysis may give some insights. The Perceptual Distinctions Approach takes time $O(|\mathcal{S}|^2)$ per step to propagate forward the state occupa-

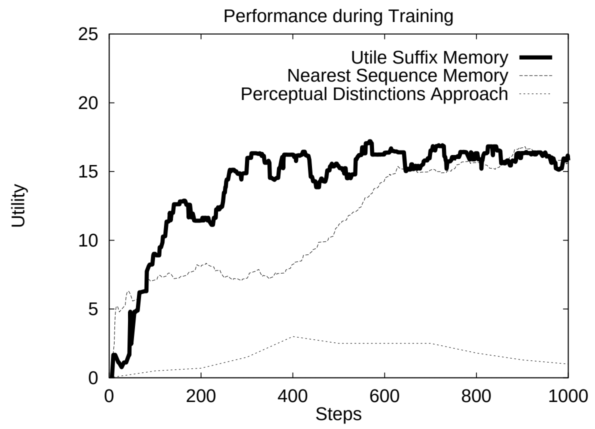


Figure 4: Performance during learning of Chrisman's spaceship docking task. The plot shows the median of ten runs.

tion probabilities, plus $O(|\mathcal{S}||\mathcal{A}|)$ to calculate Q-values; in addition, every N steps, the agent performs the Baum Welch calculation, which requires time and space $O(|\mathcal{S}|^2|\mathcal{A}|N)$, and performs $(|\mathcal{S}|^2|\mathcal{A}| + |\mathcal{S}||\mathcal{O}|)$ Chi-Square tests; in Chrisman's experiments, N is 1000. Nearest Sequence Memory requires time $O(N)$ per step, where N is the number of instances in the chain, which in these experiments was limited to 1000. USM takes time $O(|\mathcal{S}|^2|\mathcal{A}|)$ per step if the agent uses full dynamic programming (but could take much less with Prioritized Sweeping), plus time for K Kolmogorov-Smirnov tests, where K is the number of fringe nodes under a single leaf node.

An important feature of USM is that, if one is concerned about computation time, there are a number of means to gracefully reduce computation time in exchange for increased learning steps required. The use of Prioritized Sweeping and occasional Kolmogorov-Smirnov tests mentioned previously, are examples of this.

A very inefficient implementation of Utile Suffix Memory, (which does no caching of the $R(s,a)$ and $\Pr(s'|s,a)$ values for dynamic programming, does a fair amount of extraneous I/O, and does full dynamic programming and Kolmogorov-Smirnov tests at each step) learns the the spaceship docking task in 1 minute of user CPU time (2 minutes wall clock time) on a SGI workstation.

4.3 WHITEHEAD'S BLOCKS WORLD REVISITED

The agent's task is to pick and place red and blue blocks in order to uncover and lift the green block. The agent's available actions consist of *overt actions* for picking up and putting down blocks, as well as *perceptual actions*, in the form of visual routines [Ullman, 1984; Agre and Chapman, 1987], for directing its visual attention selectively. The visual-routine-manipulated region of attention is called a *marker*. The agent's perception consists of one bit that

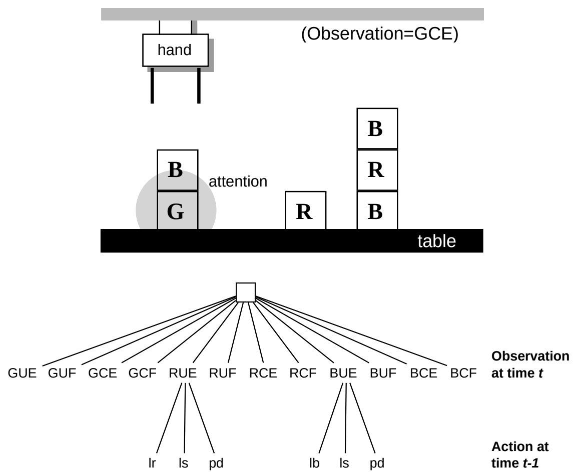


Figure 5: **Top:** A hand-eye coordination task with blocks, using visual routines. The agent must uncover the green block and pick it up. **Bottom:** The suffix tree learned by USM for the blocks task. R=red, G=green, B=blue, U=uncovered, C=covered, E=hand-empty, F=hand-full; lr=look-red, lg=look-green, lb=look-blue, lt=look-table, ls=look-top-of-stack, pu=pick-up, pd=put-down.

indicates whether the hand is empty or full, plus several bits that describe the features of the object being marked. Figure 5 (top) depicts the task environment.

The solution to this task consists of action sequences like: look-green, look-top-of-stack, pick-up, look-table, put-down, look-green, pick-up. An agent with one marker will suffer from perceptual aliasing in two states on its critical path. The state in which it is looking at the top of the stack, and the state in which it has just placed the block on the table, are each perceptually identical. Yet, in the first, the agent should next look at the table, and in the second, the agent should next look back at the green block. Because Whitehead's *Lion* algorithm can only handle hidden state by avoiding the aliased states, and because there is no way to solve this task without passing through this aliased state, Whitehead added a second marker to the agent's perception—the agent then had enough perception that there was a path from the start state to the end state the consisting solely of unaliased states. As a result of this additional marker, the agent almost doubled the number of perceptual bits, increasing the size of its perceptual state space exponentially in the number of additional bits.

Utile Suffix Memory allows us to take a very different approach. There is no need to explode the state space size throughout the task when the extra information is only needed during one step. We can start with a much smaller state space (*i.e.* only one marker), and let USM augment that state space with short-term memory only in that part of the state-space where it is needed. The result is a much smaller state space, with obvious advantages for overcoming the ubiquitous problems with the curse of dimensionality. Furthermore, we avoid the need to know ahead of time how many markers are required to perform this task without hidden state—a calculation that often requires knowing the precise sequence of actions the agent will use to solve the task.

USM was tested on this one-marker blocks task, using Whitehead's Learning by Watching technique, with a teacher intervention probability of 0.3. The teacher serves only to bias exploration, not to tell the agent which distinctions to make. Without the extra bias in this task, USM had trouble finding the sequences that lead to reward. Whitehead's *Lion* algorithm may have done better without a teacher because (1) whenever it found a state to be aliased, it never visited it again, and (2) it used a strict schedule of perceptual and overt actions (four perceptual actions, then one overt action), whereas USM does not distinguish between overt and perceptual actions. As mentioned in section 4.1, finding smarter exploration for hidden state remains an area requiring much work. USM used a temporal discount factor, $\gamma = 0.9$, a constant exploration probability $e = 0.1$, and a fringe of depth 3. The tree that solves the task, built by USM after 15 trials, is shown in figure 5 (bottom).

5 RELATED WORK

Utile Suffix Memory inherits much of its technique and desired features from UDM and NSM, but many of its ideas come from the combination of other algorithms too. Ideas from four algorithms in particular inspired the workings of USM; these are: Probabilistic Suffix Tree Learning [Ron *et al.*, 1994], Parti-game [Moore, 1993], G-algorithm [Chapman and Kaelbling, 1991] and Variable Resolution Dynamic Programming [Moore, 1991]. All four of the algorithms use trees to represent distinctions and grow the trees in order to learn finer distinctions.

Probabilistic Suffix Tree. USM has more in common with Probabilistic Suffix Tree Learning (PSTL) than any other algorithm. From it USM borrows directly the use of a tree to represent variable amounts of memory, and the use of a fringe to test for statistically significant predictions. To this common base, USM adds several features that are specific to reinforcement learning. We add a notion of actions, where layers of the tree alternately add distinctions based on previous actions and observations. We add reward to the model, and set up the statistical test to add distinctions based on future discounted reward, not observations. We also add dynamic programming in order to implement value iteration. Also note that, unlike PSTL, the training sequences are not given to USM ahead of time—USM must generate the training sequences on-line using its current suffix tree.

Parti-game. The key feature USM inherits from Parti-game is its instance-based foundation. Both algorithms take advantage of memorized instances in order to speed learning. Unlike PSTL, both USM and Parti-game must generate their instances on-line. Parti-game differs from USM in several ways. In Parti-game the agent has available a greedy controller that moves the agent towards the global goal state; new distinctions are added when the greedy controller fails. In USM, the agent has no greedy controller and works with the traditional local rewards from the environment; new distinctions are added when statistics indicate that doing so will help predict return. USM handles noisy rewards and actions well; Parti-game requires deterministic environments.

G-algorithm. From the G-algorithm, USM inherits its use of a robust statistical test on reward values, and thus, unlike Parti-game, both USM and the G-algorithm can handle noise. Unlike USM and Parti-game, the G-algorithm is not instance-based. This results in a significant inefficiency whenever a distinction is added: each time the G-algorithm splits a state, the G-algorithm is forced to reset the child state's Q-values to zero. Because the G-algorithm has no raw record of the previous experience, it cannot know how to distribute the agglomerated experience from the parent state into the new children. Each time a state is split, the G-algorithm throws away all accumulated experience for that region, and must effectively re-learn that region of state space from scratch.

Variable Resolution Dynamic Programming. Unlike PSTL, Parti-game or the G-algorithm, Variable Resolution Dynamic Programming (VRDP) and USM use dynamic programming directly on a learned model of the relevant parts of the environment. There are advantages to separating the model learning, which requires exploration, from the "non-learning" dynamic programming, which only requires computation. For instance, with such a scheme, reward can be easily propagated from crucial regions of state space that are, nevertheless, difficult to visit multiple times. In Q-learning–based algorithms, reward propagation requires multiple runs through the propagation path. Even in path play-back schemes like Experience Replay [Lin, 1991], the agent must still visit the reward state multiple times in order to propagate reward out along the multiple different paths that may be available. In addition to USM and VRDP, several other reinforcement learning algorithms take advantage of a model-based approach [Sutton, 1990; Barto *et al.*, 1991; Moore and Atkeson, 1993]. A disadvantage of VRDP is that is it partitions the state space into high resolution *everywhere* the agent visits, potentially making many irrelevant distinctions. USM, on the other hand, creates only utile distinctions.

6 DISCUSSION

Utile Suffix Memory successfully combines: instance-based learning, statistical techniques robust to noise, dynamic programming on a learned model of the environment, variable length short-memory for overcoming hidden state, and utile memory distinctions. The algorithm seems to fit many disparate technical ideas together smoothly, and the preliminary experimental results look promising.

There are several interesting possibilities for further development: (1) The tree could hold perceptual distinctions as well as history distinctions, forming a framework for generalized "utile distinctions." (2) The fringe branches could test more sophisticated distinctions by using complex tests supplied by the agent designer, thus providing a straightforward means for imparting domain knowledge to the agent. (3) We could modify the tree to make utile distinctions in *continuous* perception and action spaces. (4) We could handle noisy perception better by explicitly representing it using techniques from probabilistic decision trees, or, better yet, applying the lessons of USM to learning partially observable Markov decision processes. For more detailed explanation and discussion of these ideas, see [McCallum, 1994].

Acknowledgments

This work has benefited from discussions with many colleagues, including: Dana Ballard, Andrew Moore, and Jonas Karlsson. This material is based upon work supported by NSF under Grant no. IRI-8903582 and by NIH/PHS under Grant no. 1 R24 RR06853-02.

References

[Agre and Chapman, 1987] Philip E. Agre and David Chapman. Pengi: an implementation of a theory of activity. In *AAAI*, pages 268–272, 1987.

[Barto *et al.*, 1991] A.B. Barto, S.J. Bradtke, and S.P. Singh. Real-time learning and control using asynchronous dynamic programming. Technical Report 91-57, University of Massachusetts, Amherst, MA, 1991.

[Bellman, 1957] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[Bertsekas and Shreve, 1978] Dimitri. P. Bertsekas and Steven E. Shreve. *Stochastic Optimal Control*. Academic Press, 1978.

[Cassandra *et al.*, 1994] Anthony R. Cassandra, Leslie Pack Kaelbling, and Michael L. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA, 1994.

[Chapman and Kaelbling, 1991]
David Chapman and Leslie Pack Kaelbling. Learning from delayed reinforcement in a complex domain. In *Twelfth International Joint Conference on Artificial Intelligence*, 1991.

[Chrisman, 1992] Lonnie Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Tenth National Conference on AI*, 1992.

[Jaakkola *et al.*, 1995] Tommi Jaakkola, Satinder Pal Singh, and Michael I. Jordan. Reinforcement learning algorithm for partially observable markov decision problems. In *Advances of Neural Information Processing Systems (NIPS 7)*. Morgan Kaufmann, 1995.

[Lin, 1991] Long-Ji Lin. Programming robots using reinforcement learning and teaching. *Ninth National Conference on Artificial Intelligence*, 1991.

[Lin, 1993] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1993.

[Littman, 1994] Michael Littman. Memoryless policies: Theoretical limitations and practical results. In *Proceedings of the Third International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, 1994.

[McCallum, 1993] R. Andrew McCallum. Overcoming incomplete perception with utile distinction memory. In *The Proceedings of the Tenth International Machine Learning Conference*. Morgan Kaufmann Publishers, Inc., 1993.

[McCallum, 1994] R. Andrew McCallum. Utile suffix memory for reinforcement learning with hidden state. Technical Report 549, University of Rochester Computer Science Dept., December 1994.

[McCallum, 1995] R. Andrew McCallum. Instance-based state identification for reinforcement learning. In *Advances of Neural Information Processing Systems (NIPS 7)*, 1995.

[Moore and Atkeson, 1993] Andrew W. Moore and Christopher G. Atkeson. Memory-based reinforcement learning: Efficient computation with prioritized sweeping. In *Advances of Neural Information Processing Systems (NIPS 5)*. Morgan Kaufmann Publishers, Inc., 1993.

[Moore, 1991] Andrew W. Moore. Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. *Proceedings of the Eighth International Workshop on Machine Learning*, pages 333–337, 1991.

[Moore, 1993] Andrew W. Moore. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state spaces. In *Advances of Neural Information Processing Systems (NIPS 6)*, pages 711–718. Morgan Kaufmann, 1993.

[Peng and Williams, 1992] Jing Peng and R. J. Williams. Efficient learning and planning within the Dyna framework. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, 1992.

[Platzman, 1977] Loren Kerry Platzman. *Finite Memory Estimation and Control of Finite Probabilistic Systems*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, January 1977.

[Ron *et al.*, 1994] Dana Ron, Yoram Singer, and Naftali Tishby. Learning probabilistic automata with variable memory length. In *Proceedings Computational Learning Theory*. Morgan Kaufmann Publishers, Inc., 1994.

[Sutton, 1990] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, June 1990.

[Teller, 1994] Astro Teller. The evolution of mental models. In Kim Kinnear, editor, *Advances in Genetic Programming*, chapter 9. MIT Press, 1994.

[Thrun, 1992] Sebastian B. Thrun. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, CMU Comp. Sci. Dept., January 1992.

[Ullman, 1984] Shimon Ullman. Visual routines. *Cognition*, 18:97–159, 1984.

[Watkins, 1989] Chris Watkins. *Learning from delayed rewards*. PhD thesis, Cambridge University, 1989.

[White and Scherer, 1994] Chelsea C. White and William T. Scherer. Finite-memory suboptimal design for partially observed markov decision processes. *Operations Research*, 42:439–455, 1994.

[Whitehead and Ballard, 1991] Steven D. Whitehead and Dana H. Ballard. Learning to perceive and act by trial and error. *Machine Learning*, 7(1):45–83, 1991.