Utrecht University
Institute of Information and Computing Sciences

Statistical Models for Non-Markovian Control Tasks

Daan Wierstra

February, 2004

Supervisor: Dr. M.A. Wiering
Co-supervisor: Prof. Dr J.-J. Meyer
INF/SCR-04-06

Contents

1 Introduction **1**

2 POMDPs and Memory: A Very Short Introduction **4**
- 2.1 Introduction 4
- 2.2 Markov Decision Processes 5
- 2.3 Reinforcement Learning 7
  - 2.3.1 Exploration 7
  - 2.3.2 Eligibility Traces 8
- 2.4 Partially Observable Markov Decision Processes 9
- 2.5 POMDP Approaches 9
  - 2.5.1 Approximations with a Model 10
  - 2.5.2 Memory and Utile Distinction 11
  - 2.5.3 Other Methods 11
- 2.6 POMDPs and Memory: an Overview 12
- 2.7 Discussion 14

3 Utile Distinction Hidden Markov Models **15**
- 3.1 Introduction 15
- 3.2 Utile Distinction and HMMs 16
- 3.3 Algorithm Details: Utile Distinction Hidden Markov Models . . . 17
- 3.4 Algorithm Details: Coupled UDHMMs 20
- 3.5 Experimental Results 21
  - 3.5.1 Hallway Navigation 21
  - 3.5.2 Detecting Long-Term Dependencies: The T-Maze 23
  - 3.5.3 The 89-State Maze 24
- 3.6 Discussion 26

4 Hierarchical Methods in Reinforcement Learning **28**
- 4.1 Introduction 28
- 4.2 Hierarchical Methods: an Overview 29
- 4.3 Automatic Discovery of Hierarchical Structure 31
- 4.4 Discussion 33

5 A New Implementation of Hierarchical Hidden Markov Models **34**
- 5.1 Introduction 34
- 5.2 The Three Problems 35
- 5.3 A new HHMM algorithm 36
- 5.4 Discussion 38

6 Model Operators and Hierarchical UDHMMs 39
6.1 Introduction 39
6.2 A Hierarchical Behavior Model: HUDHMM 41
6.3 The Hierarchical Model Operators (HMO) Algorithm 42
6.3.1 The Observation Split Operator 42
6.3.2 The Brother Split Operator 43
6.3.3 The Uncle Split Operator 44
6.3.4 The Parent Split Operator 45
6.3.5 The Cousin Split Operator 45
6.4 Experimental Results 47
6.4.1 Detecting Long-Term Dependencies: The T-Maze 47
6.4.2 The 89-State Maze 48
6.5 Discussion 48

7 Conclusion 50

Chapter 1

Introduction

Consider the problem of an explorer robot navigating the surface of the planet Mars. The robot moves about and makes local observations. Since no human operators are available to steer the machine in real-time, the robot must decide for itself what to do given the circumstances, the 'circumstances' being its history of observations and previously taken actions. It cannot disambiguate its current situation and position from the current observation alone, because different situations might produce the same observations. This robot has to deal with the problem of *hidden state* (Lin & Mitchell, 1993). It has to disambiguate its hidden state so much as to find a good policy to deal with its environment. We say it has to deal with *perceptual aliasing* (Whitehead & Ballard, 1991). We generally refer to this sort of problem as a Partially Observable Markov Decision Process or POMDP (Sondik, 1978).

Perceptual aliasing can be viewed as both a blessing and a curse. It is a blessing, because it gives the agent the opportunity to generalize across states — different states that nevertheless produce the same or similar observations are likely to require the same actions. It is a curse, because, for optimal action selection, the agent needs some kind of memory. That is a very serious problem indeed, because, in order to create memory, it has to keep track of its history of actions and observations to make a good estimate of its current world state. Unexpected events, malfunctioning motors and noisy observations make the problem even more difficult. How can this problem possibly be solved?

Preprogramming all actions for all possible situations is not always an option. It does not solve the problem of ambiguous observations, and, furthermore, a correct 'world model' is not always available to the programmer. Also, often the number of possible situations is much larger than a programmer could possibly anticipate. Moreover, the designer does not always know *what* exactly the optimal actions are.

So, we conclude, for this class of problems, learning algorithms are essential. More specifically, Reinforcement Learning (Watkins, 1989; Kaelbling, Littman & Moore, 1996; Sutton & Barto, 1998) algorithms. In Reinforcement Learning (RL), an agent observes the state of the world and acts accordingly as to maximize long-term reward. The reward is the teaching (reinforcement) signal that is given to the agent when it reaches favorable states or 'goal' states. In the beginning, the agent acts randomly, not knowing what actions are 'right' and which are 'wrong'. But after a while, it learns to behave more rationally

because of the reinforcements (rewards and punishments) carefully provided by the teacher. When the agent's behavior is satisfactory, the agent can be put to practice.

Several Reinforcement Learning algorithms have been developed, among which the Q-learning (Watkins, 1989) framework is one of the most important. The problem with Q-learning is that the agent needs to know the state of the world in order to be able to act optimally. It cannot deal with perceptual aliasing: different states that nevertheless produce the same observations. It cannot act optimally in a world that is only partially observable. We therefore need algorithms that keep track of history. POMDPs — Partially Observable Markov Decision Processes — provide a useful framework for that.

POMDP algorithms go further than basic Reinforcement Learning. In the POMDP framework, an agent is supposed not only to maximize discounted long-term reward (as in standard RL), but also to simultaneously solve the incomplete perception problem: it must observe and act in a world, while its observations of the world are incomplete. It only 'sees' partial states, because the real world state is hidden. In order to behave optimally, the agent must therefore build some kind of internal 'behavior model' in order to solve the problem of partial observability. In other words, it must learn to keep track of a 'hidden internal state'.

For this class of problems three approaches exist: the optimal approach, and the heuristic method provided with a world model, and the heuristic method without any world model (the 'memory approach'). The optimal approach seeks to solve the POMDP problem exactly, that is, exact algorithms provide results that are provably optimal. However, POMDP problems tend to be extremely hard to solve, and exact solutions are computationally infeasible for most real-life problems.

Therefore, heuristic algorithms are needed. Problems that are provided with a world model (a model that describes what world states there are, what the effect of actions taken by the agent on those states is, and what observations to expect for each of those states) are naturally much easier to solve than cases where no model exists. But the availability of a world model is also somewhat unrealistic for many real-life problems: we don't have a detailed map of the surface of Mars, for example, and we don't know the exact location of every little rock, crack or hill there. Moreover, the world constantly changes, and a static world model would not do our robot any good if the world did not turn out to be like its model.

In this thesis we concentrate on heuristic memory-based POMDP algorithms that let the agent build its own internal behavior model 'on the fly' during the learning process, such that a user-provided world model is not required anymore. We present three novel POMDP algorithms, and show their performance on several POMDP problem domains. For one very hard, very stochastic problem domain, the 89-state Maze, we show a superior performance of one of the algorithms as compared to other alternatives known to us. Furthermore, we develop a new Expectation-Maximization (EM) algorithm (Dempster, Laird & Rubin, 1977) that aids the hierarchical approach to solving very stochastic POMDPs.

In chapter 2, we first present a short introduction to Reinforcement Learning and establish the necessary notation and formalisms necessary for understanding the POMDP framework. We give an overview of existing POMDP algorithms that have inspired our approaches. In chapter 3, we present a description of our

algorithms Utile Distinction Hidden Markov Models (UDHMM) and Coupled Utile Distinction Hidden Markov Models (CUDHMM), along with experimental results on various problem domains. Both (strongly related) algorithms build on detecting utile distinction in POMDP environments with the use of HMMs and Coupled HMMs, respectively.

In chapter 4, we present a general overview of hierarchical methods in Reinforcement Learning, with special interest for, but not exclusively focusing on the relation to POMDPs.

In chapter 5, we develop a new — and simple — EM algorithm for dealing with Hierarchical Hidden Markov Models (HHMM) (Fine, Singer & Tishby, 1998), that reduces the time complexity of model re-estimation from $\mathcal{O}(NT^3)$ to $\mathcal{O}(N^2T)$. This HHMM will in chapter 6 be used as memory model for Hierarchical Utile Distinction Hidden Markov Models (HUDHMM), a logical extension of UDHMMs to the hierarchical case. In order to take full advantage of the hierarchical structure, we develop the Hierarchical Model Operator (HMO) algorithm, that operates on hierarchical models in order to improve their POMDP performance. Together, HMO and HUDHMM show a very good performance on the 89-state Maze task.

The novel contributions of this thesis are the UDHMM and CUDHMM algorithms, the *simple* linear-time inference algorithm for HHMMs, and HMO-HUDHMM.

Chapter 2

POMDPs and Memory: A Very Short Introduction

2.1 Introduction

Partially Observable Markov Decision Processes (Sondik, 1978) provide a useful formal framework for reasoning about agents that learn policies from delayed rewards, and act in environments that are both stochastic and only *partially* observable. With 'partially observable' we mean that the state of the world is not known to the agent. Rather, it *senses* or perceives observations instead of perceiving the direct world state. In this uncertain environment the agent must act, initially not knowing which policies are optimal, but gradually learning from — possibly delayed — rewards or reinforcement signals from a reward function that the teacher (programmer) provides.

The advantages of this reward attribution scheme are obvious: the human programmer need not bother writing complex programs anymore, but rather he can specify a set of 'goals' for which a positive reinforcement is defined. The idea is that then the agent can figure out for itself, through trial-and-error, what the optimal policies are. The field of Reinforcement Learning (RL) (Watkins, 1989; Sutton & Barto, 1998) concerns itself with research on this topic.

However, most research in RL focuses on the completely observable case, where the agent perceives not observations but the state of the world directly. Most RL algorithms are designed for the more usual class of Markov Decision Process (MDP) problems, where knowledge of the current state is provided to the agent, which is completely informative. Since this can be a severe limitation in real-world tasks, recent research interest has turned to POMDPs, a more powerful concept than MDPs, where the notion of observation instead of state is formalized.

The purpose of this chapter is to provide a short introduction to POMDPs and a formal definition of the POMDP paradigm. We introduce successively MDPs and some basic RL techniques, which are necessary to understanding the context, and the POMDP formalism. We discuss several approaches to solving POMDPs, but we focus on the *model-free* case. In a model-free POMDP, not only the true state of the world is unknown to the agent, but not even a world model is provided to it. Basically, it has to learn everything from scratch, only

relying on experience from its own actions, its perceptions and rewards. It must develop some sort of memory to cope with hidden world state and delayed rewards.

Model-free POMDPs are a very hard problem. Since finding optimal solutions to the problem is so hard, we can say that heuristic solutions are appropriate. Still, very few completely model-free heuristic algorithms exist today. Many approaches that do exist assume at least some domain knowledge to be incorporated in a solution method. We, however, focus on the completely model-free case, which is theoretically more interesting but also more challenging. In this short overview we present some methods that are applicable to the completely model-free case.

2.2 Markov Decision Processes

A Markov Decision Process (MDP) is a formalism that describes the interactions of an agent with the world. It consists of a number of world states the agent can be in, a number of actions the agent can perform plus the state-transition probabilities for every action, and a reward-function. An MDP can be formally described as a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, T, R \rangle$ where

- $\mathcal{S}$ is a set $\{s_1, s_2, \ldots, s_N\}$ of world states
- $\mathcal{A}$ is a set $\{a_1, a_2, \ldots, a_L\}$ of actions
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is a probabilistic state-transition function. For every state, action, and possible successor state, $T(s, a, s')$ denotes the probability of ending in the successor state s' given the start state s and action a.
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a (possibly stochastic) local reward function. For every state and action, $R(s, a)$ gives a local reward for executing action a in state s, according to a probability distribution over real values.

The MDP produces, at every time step $t = 1, 2, 3, \ldots$, a state s_t after execution of action a_{t-1} in state s_{t-1}. Additionally, a *reward* r_t is given at every time step, after the reward function R. Associated with an MDP is the *policy* $\pi(s, a)$, that describes the agent's current policy for every state. $\pi(s, a)$ denotes the probability of selecting action a in state s.

We consider MDP problems with an *infinite horizon, discounted* model, that is, problems with a possibly infinite time horizon and where the agent tries to maximize the expected long-term discounted reward. Thus, the problem for such an MDP is to find an optimal policy π^* that maximizes the expected long-term discounted reward,

$$\mathcal{E}\Big[\sum_{t=0}^{\infty} \gamma^t r_t\Big]$$

where $\gamma \in (0, 1]$ is the discount factor over time.

An important notion in MDP theory is the *Markov* property. We assume that in our MDP, states are *Markov*, i.e., consistent and stationary. When a state is Markov, the next state is only dependent on the current state of the

world and the action taken from there. It does not depend on other historical information. In other words, the state gives a complete description of the agent's situation in the world. More formally

$$Pr(s_{t+1} = s|s_t, a_t) = Pr(s_{t+1} = s|s_t, a_t, \ldots, s_1, a_1)$$

For all systems where this equation holds, we say the Markov property holds. In many Reinforcement Learning techniques (see below), for example, we assume the Markov property.

Now we define the *value* $V^\pi(s)$ of a state s in this MDP with policy π to be the expected discounted reward (also called *return*) received from starting in state s and following policy π from there afterwards.

$$\begin{aligned} V^\pi(s) &= \mathcal{E}[r_t + \gamma r_{t+1} + \gamma r_{t+2}^2 + \cdots |s_t = s, \pi] \\ &= \mathcal{E}\Big[\sum_{i=t}^{\infty} \gamma^{i-t} r_i | s_t = s, \pi\Big] \end{aligned}$$

Likewise, for every state-action pair (s, a) we define the *quality* $Q^\pi(s, a)$ to be

$$\begin{aligned} Q^\pi(s,a) &= \mathcal{E}[r_t + \gamma r_{t+1} + \gamma r_{t+2}^2 + \cdots |s_t = s, a_t = a, \pi] \\ &= \mathcal{E}\Big[\sum_{i=t}^{\infty} \gamma^{i-t} r_i | s_t = s, a_t = a, \pi\Big] \end{aligned}$$

We assume here that policy π greedily selects actions at every state for which $Q(s, a)$ is highhest. More formally, following from our MDP definition and problem formulation, we can write the so-called *Bellman equations* (Bellman, 1961) for arbitrary policy π as

$$V^\pi(s) = \sum_a \pi(s,a)(\mathcal{E}[R(s,a)] + \gamma \sum_{s'} T(s,a,s') V^\pi(s'))$$

for all states $s \in \mathcal{S}$, and

$$Q^\pi(s,a) = \mathcal{E}[R(s,a)] + \gamma \sum_{s'} T(s,a,s') \sum_{a'} \pi(s',a') Q^\pi(s',a')$$

for all states $s \in \mathcal{S}$ and actions $a \in \mathcal{A}$.

Given a certain MDP, we want to find an optimal or near-optimal policy. In the optimal case, the agent tries to maximize its expected return. We write the equations for *optimal* value and quality functions $V^*(s)$ and $Q^*(s, a)$ as

$$V^*(s) = \max_a(\mathcal{E}[R(s,a)] + \gamma \sum_{s'} T(s,a,s') V^*(s'))$$

and

$$Q^*(s,a) = \mathcal{E}[R(s,a)] + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q^*(s',a').$$

2.3 Reinforcement Learning

In Reinforcement Learning, one is concerned with designing an efficient algorithm for finding increasingly good policies in MDPs. One algorithm that is often used is called Q-learning (Watkins & Dayan, 1992). Q-learning functions completely online. It starts with random or zero-initialized Q-values, that increase in quality during the agent's execution in the world. At every time step t, the agent performs an action a in state s_t according to its Q-value at that time step. It usually selects the best possible action, but occasionally performs an *exploratory* step in order to visit parts of state space it would otherwise, with a still very imperfect policy, maybe never reach. The Q-update rule is as follows:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r_t + \gamma \max_{a'} Q(s',a')),$$

where $\alpha \in (0,1]$ denotes the learning rate of the algorithm.

Using this update rule, the policy initially starts out behaving very poorly, but it gradually gets better and better. It can be shown that under certain conditions where α is appropriately decreased during time and every action is executed infinitely often in every state, Q-learning converges with probability 1 to the optimal values Q^* for an optimal policy π^* (Watkins & Dayan, 1992; Tsitsiklis, 1994).

Another Q-update rule that is often used is called *SARSA-learning*, after the experience tuple $\langle s_t, a_t, r_t, s_{t+1}, a_{t+1} \rangle$ that is made available to the agent at every time step. The only difference with Q-learning is that the max-over-actions expression on the right hand side is replaced by the Q-value of the next state-action pair:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r_t + \gamma Q(s',a'))$$

Of course, Dynamic Programming (DP) methods could use the Bellman equations shown above for the given MDP to approximate an optimal policy. This can be done as follows. During *online* experience trials, the agent estimates a model for transitions between states $T(s,a,s')$ and return distributions for every state $R(s,a)$. After a sufficient estimation of this model has been constructed, we can apply DP *offline*, that is, after the trials, on the above equations to successively update better estimations $Q^{(k)}, Q^{(k+1)}, \ldots$. However, this method is generally very slow. Moreover, aside from being computationally expensive, it includes inconvenient offline phases for which an increasingly good model must be available. Because of that, several more efficient Reinforcement Learning techniques such as the above have been developed that perform *online* updates of state-action values $Q(s,a)$ without the need for an offline phase.

2.3.1 Exploration

Of course, if the agent, biased by bad initial Q-values, never reaches certain important parts of state space, it will never develop an optimal policy. Hence we must ensure the agent takes enough exploratory actions. On the other hand, if the agent takes too many exploratory actions, it might never reach the best parts of state space either (its goals) since its actions are so random. Here we speak of an exploration/exploitation trade-off.

An often used exploration technique is to have, at every time step, a certain chance ϵ (say, 0.1) that the agent performs a random action. It is usual to let ϵ slowly decrease over time. Applying this exploration method, we make sure that the agent, in the limit, visits every state infinitely often, and performs every action at every state an infinite number of times, thereby satisfying the convergence criterion.

Another technique, called *Boltzmann exploration*, selects actions randomly but 'weighted' according to their estimated Q-values. If a Q-value of an action in a certain state is higher than the Q-values of other actions, it is selected with higher probability. This method includes a *temperature* τ which we usually decrease over time. Temperature indicates the volatility of action-selection. The effect is that initially, most actions have about equal probability of being selected, but later, when Q-values are more 'certain' — that is, better estimates of Q^* — the action-selection policy tends to look more and more like a greedy max-function. The philosophy behind this is that after a while, when Q-values have been updated a number of times, selecting actions with higher Q-values tends to lead to better opportunities to *exploit* parts of state space that are probably advantageous. This should, hopefully, lead to faster convergence to the optimal policy.

For Boltzmann exploration, the action selection probabilities for action a in state s are defined as:

$$\pi(s,a) \;=\; \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}}$$

2.3.2 Eligibility Traces

Another issue with regard to Reinforcement Learning techniques we will consider, is that of *eligibility traces*. This technique is inspired by the following problem. Imagine an agent acting in an environment with only zero-reward states but for one goal state with a reward. When the agent reaches the goal for the first time, only one update is made, namely for the state visited immediately before the goal state. However, it would be more efficient to update the state before that as well, and the state before that, and so on. Otherwise, reaching an optimal policy for reaching a goal that is 10 steps away from the start state could require the agent almost 10 times reaching that goal in just so many trials.

A solution comes from eligibility traces. The eligibility of a state-action pair is the degree to which that pair has been experienced in the recent past, moderated by parameter $\lambda \in [0,1]$. For every update on the current state action pair, we now also update *every* state-action pair according to their eligibility. Q-learning and SARSA-learning with eligibility traces are called Q(λ)-learning and SARSA(λ)-learning.

An example of the application of eligibility traces, SARSA(λ)-learning, is illustrated below:

SARSA(λ)-learning for $\langle s_t, a_t, r_t, s_{t+1}, a_{t+1}\rangle$:
1) $\delta_t \leftarrow r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$
2) For each state-action pair (s, a) do:
  2a) $e_t(s, a) \leftarrow \gamma\lambda e_{t-1}(s, a)$
  2b) $Q(s, a) \leftarrow Q(s, a) + \alpha\delta_t e_t(s, a)$
3) $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta_t$
4) $e_t(s_t, a_t) \leftarrow e_t(s_t, a_t) + 1$

2.4 Partially Observable Markov Decision Processes

Now we turn to the formal definition of POMDPs. A Partially Observable Markov Decision Process $\mathcal{P} = \langle \mathcal{M}, \mathcal{O}, \mathcal{Z}\rangle$ consists of

- An MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, T, R\rangle$,
- A set of possible observations $\mathcal{O}$, where $\mathcal{O}$ could constitute either a set of discrete observations or a set of real-valued observation vectors,
- $\mathcal{Z}$, a probability density mapping state-observation combinations $\mathcal{S} \times \mathcal{O}$ to a probability density distribution, or, in the case of discrete observations, a probability function mapping state-observation combinations $\mathcal{S} \times \mathcal{O}$ to probabilities. In other words, $\mathcal{Z}(s, o)$ yields the probability of observing o in state s.

So basically, a POMDP is like an MDP but with observations instead of direct state perception.

2.5 POMDP Approaches

The states $\mathcal{S}$ in a POMDP are now called *hidden states* instead of states, since the agent does not directly perceive world state anymore, but instead observes observations. If a world model (which includes the fact which hidden states $\mathcal{S}$ there are in the model, transition probabilities T between them, and observation probabilities $\mathcal{Z}$ per hidden state) is available to the agent, it can easily calculate and update a *belief vector* $\vec{b}_t = \langle b_t(s_1), b_t(s_2), \ldots, b_t(s_N)\rangle$ over hidden states at every time step t by taking into account the history trace $h = \{o_1, a_1, o_2, \ldots, a_{t-1}, o_t\}$ — a matter of simple likelihood calculation.

However, even with this model, it is not trivial to compute an optimal policy. Look at Figure 2.1 to get an idea of the complexity of the task. Imagine a world where you are the president of the United States, and you must decide whether to attack a certain Arab country that is suspected of having Weapons of Mass Destruction (WMDs), but that itself claims otherwise. You have three possible options: make peace with the country, which yields great positive reinforcement if the country indeed did not have WMDs, but a nuclear attack otherwise. You can send in weapons inspectors to update your belief on the WMD possession — but be careful, since their observations are noisy and often incorrect. Or you can attack, which yields positive reinforcement if indeed WMDs were found, but negative reinforcement (no re-election) if they were not there.

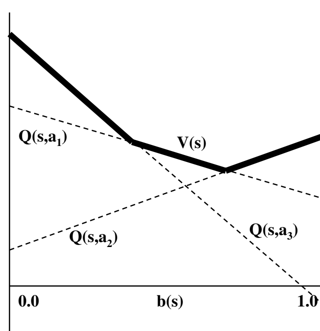


Figure 2.1: Hypothetical value function in belief space for a 2-state POMDP, the WMD case. The outer left of the graph yields the value when the belief is 100% that the state is s_1 (no WMDs present), the outer right indicates the value when the belief is 100% that the state is s_2 (WMDs present), while the middle indicates the belief for the 50%–50% case. The values of three actions are shown, each being the best one in at least a part of belief space. Action a_1 (more inspections) has the highest value when uncertainty about the WMDs is great. When WMDs are probably present (right part of the graph), action a_2 (attack) is the optimal action. On the other hand, when evidence shows that there are probably no WMDs, making peace (action a_3) is probably the best thing to do.

The most sensible strategy, when certain of WMDs, is to attack. When certain that they are not there, we make peace. But what of cases in-between? What if our belief is 50%–50%? Then we send in the inspectors, until our certainty has reached a sufficient level to change the policy. Figure 2.1 shows a 2-state, 3-action POMDP with a similar structure.

It should be clear that constructing an exact, optimal solution for more complex value functions for more than these two states can quickly become very difficult. A simple, exact solution is given by Sondik's (1971) One-Pass algorithm. A better algorithm, that more efficiently deals with so-called 'sunken vectors', is the Witness Algorithm (Cassandra, Kaelbling & Littman, 1994). Even more sophisticated is Incremental Pruning (Zhang & Liu, 1996). However, optimal solutions remain computationally infeasible beyond several dozen hidden states. This forces us to consider *approximate* algorithms.

2.5.1 Approximations with a Model

We cannot compute values for the entire belief space when tasks get bigger than several dozen hidden states. So, for larger problems, approximations are needed to solve this issue. One early approach is called *linear Q-learning* (Littman, Cassandra & Kaelbling, 1995), which, in combination with some initialization heuristic, produces reasonable results on certain problem domains, including a 89-state maze with very noisy actions, perceptions, and rewards. The basic idea of linear Q-learning is to use the belief vector as an input to a kind of 2-layer linear network where the target Q-value is approximated by considering the belief the 'input' and the q-values (notice the small caps) the 'weights' in the

network:

$$\triangle q_a(s) = \alpha b(s)(r + \gamma \max_{a'} Q_{a'}(\vec{b}') - \vec{q}_a \cdot \vec{b}).$$

where $Q_a(b)$ is calculated by a 'voting' heuristic $Q_a(\vec{b}) = \vec{q}_a \cdot \vec{b}$.

Other techniques include, for example, various grid methods (Hauskrecht, 2000), that partition the continuous belief space with grid points. Deciding where to put those grid points and how to interpolate between them are the key issues for these algorithms. An approach aimed an solving POMDPs with continuous state spaces and action spaces is given by Thrun (2000). But all these approaches fall beyond the scope of this paper, since we focus on POMDPs *without* any model.

2.5.2 Memory and Utile Distinction

When a world model is not available, when we do not have access to a belief vector indicating our assessment of the situation, we need to turn to other measures. In particular, the notions of *memory* and *utile distinction* spring to mind.

The agent needs memory, since perceptual aliasing has to be dealt with. If different situations that need different actions but are nevertheless producing identical or similar observations are encountered, some event or sequence of events in history should be recognized as indicative of hidden state discrimination and stored in memory such that the agent can deal with the situation. Therefore, memory is necessary in model-free POMDPs.

The agent needs to base its discrimination and creation of memory capacity not only on observations, but also on utile distinction — the rewards received. Situations with similar observations but wildly different reward characteristics can be different, and must somehow be discriminated.

A way of creating new memory capacity, preferably only when needed so that the memory bookkeeping does not slow down too much, must be found. We call the memory construction method plus the created memory plus the developed policies based on this an *anticipatory behavior model.* It is not a world model, since we need not model the entire world if the task is simple enough. Neither is it just a policy, since action selection must be based on both perceptual and utile distinctions. It is an anticipatory behavior model, that describes what to do in which situation, where a situation is not a world state but an observation and possibly memory describing past events.

One more thing needs to be said about model-free POMDPs: often, in many real-world tasks, a *memory-less* policy can do very well as well. For example, Loch and Singh (1998) show very good results with SARSA(λ)-learning on a number of tasks. They replaced states s in the Q-tables with observations o, showing that purely reactive policies can do well on many occasions.

2.5.3 Other Methods

Several other POMDP solution methods have been developed that are not or only vaguely related to the approaches we will present in this thesis. One class of solutions is based on neural network architectures. Recurrent neural networks, such as Lin and Mitchell's (1992) Recurrent-Q, solve the partial observability

problem by feeding history information into the network at every time step. This algorithm has only a limited history window, though. Long Short-Term Memory neural networks applied to Reinforcement Learning (RL-LSTM) (Bakker, 2004) offer a solution to that by using hidden *gating* neurons that can hold memory for much longer time periods.

A good neuroevolutionary approach by Gomez and Miikulainen (1999), Enforced Sub-Populations, evolves hidden neurons of a neural net independently to teach the network the control task. Lanzi (1997) presents a method for evolving classifiers that deal with internal memory. Aberdeen (2002) provides yet another take on POMDP problems, that of *policy-gradient* algorithms. Many more algorithms have been developed, but we want to concentrate on truly model-free approaches, as sketched below.

2.6 POMDPs and Memory: an Overview

Here we give a brief overview of some truly model-free POMDP approaches that have been developed so far.

The Perceptual Distinctions Approach Chrisman (1992) proposed the Perceptual Distinctions Approach, using a Hidden Markov Model (Rabiner, 1989) for memory creation. In a Hidden Markov Model (HMM), there are a number of hidden states, transitions between those, and observation models for all states. The HMM re-estimation algorithm, using training data (sequences of observations), estimates these model parameters as well as possible, such that the probability likelihood of the training data is maximized in the re-estimated model. HMMs are often used in applications such as speech recognition.

Chrisman proposed modifying an HMM with transition probabilities conditioned by the possible actions. Chrisman's agent is fed with histories of observations and actions, and the HMM re-estimation algorithm optimizes the agent's perceptual model of the world. On top of this, a Q-learning technique similar to the one discussed above is superimposed, imagining a 'belief' vector over the HMM's hidden states. This results in an agent that can cope, to a certain extent, with perceptual aliasing. The agent can solve several simple POMDP tasks.

However, this method lacks a method for utile distinction, which renders the agent unable to cope with many tricky situations that rely on utility discriminations in hidden state space.

Utile Distinction Memory McCallum's Utile Distinction Memory (UDM) (McCallum, 1993) builds on Chrisman's Perceptual Distinctions Approach, but enhances it by introducing state splits for utile distinction. In order to do that, it keeps return (future discounted reward) statistics per node. UDM performs statistical tests whether splitting states would help predicting utility. If a node has two wildly different return *distributions* depending on prior activation of two earlier nodes, the node is split and transition probabilities are divided between the two condition nodes. This can greatly help predicting return, enabling the algorithm to deal with more complex POMDP tasks than Chrisman's algorithm.

However, splitting states is only done by considering the previous state, only one time step back in history. This makes many POMDP tasks that depend

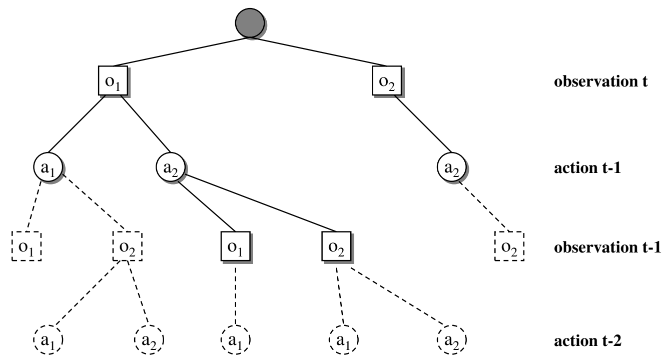


Figure 2.2: A Utile Suffix Memory tree structure. The tree models experience along its branches, every level deeper in the tree hierarchy indicating one time step further away. Square nodes indicate observations, round nodes actions. The dashed parts of the tree are the so-called 'fringe nodes' that are not actually used but can become real nodes when a statistical test shows the advantages thereof.

on recognizing events longer back in history insoluble to UDM. Also, event sequences that are, as a sequence, indicative of expected return, cannot be discriminated by UDM. This problem is solved by McCallum's next algorithm (McCallum, 1995a), USM, at a cost however of losing the explicit probabilistic framework.

Utile Suffix Memory and U-tree Utile Suffix Memory (McCallum, 1995a) and U-tree (McCallum, 1995b) constitute approaches based on a decision tree with Q-learning superimposed. Every node in the tree is either an action or an observation (see Figure 2.2). To arrive at a USM 'hidden state', start at the root and go down along the branches of the tree, where the current observation is the first choice, the last action performed the choice after that, and so on, until you arrive at a leaf node or cannot go any further. At the node where you arrive, Q-values for the different actions are stored so that you can choose an action.

USM and U-tree are instance based algorithms, where every experience is stored as an instance somewhere along the branches of the tree. The deepest parts of the tree are the 'fringe nodes', nodes that do not really count as nodes, but that are changed to real nodes whenever a statistical test on utile distinction (similar to UDM) shows that splitting the non-fringe parent node into the fringe nodes could provide a statistically significant advantage to utility prediction.

U-tree includes a method for selective perception, in cases where the observations are made up of several elements but where many of those elements of an observation might not be necessary for utility prediction enhancement at all. Selective perception enables the agent to deal with even more complex environments than USM already can handle. Those algorithms perform generally very well. Since they rely on decision trees, the main problem is that they are not very well suited to stochastic environments, or environments where observations can be real-valued vectors, for example. Also, longer time dependencies cannot be dealt with by USM or U-tree.

HQ-learning HQ-learning, as introduced by Wiering and Schmidhuber (1997), is an algorithm that can handle long-term time dependencies by introducing a successive hierarchy of different agents $A_1 \ldots A_N$ that learn to hold or give up activation to the next agent. The first agent A_1 starts out, selecting actions from an ordinary Q-table but with states replaced by observations. However, not only are the agents selecting actions, they are also, at every time step, contemplating giving over control to the next agent. They do so by a method similar to Q-learning, but considering the expected value of the next agent (taken from the Q-table of the next agent) compared with the expected value of its own continued activation.

HQ-learning manages to solve large tasks that could never be handled by algorithms like U-tree or UDM. However, HQ is much more vulnerable to noisy environments, since one noisy observation could yield activation to the next agent when it should not.

Self-Segmentation of Sequences (SSS) The same can be said for SSS (Sun and Sessions, 2000). However, this approach is more flexible than HQ. It basically extends the basic HQ structure by introducing a true tree-formed hierarchy of agents, where higher agents can 'call' lower-level agents as if they were actions. The stopping criterion, the moment when control should be given back to an agent's parent, is learned much the same way as HQ's learning to give up control to the next agent in line.

The great advantage of a hierarchy like this is that exploration strategies tend to become much more efficient. If, in a large environment, an agent performs an exploratory action, it still remains in the same region of hidden state space. While, on the other hand, a hierarchical exploratory action could 'test' the combinations of whole chunks of policies that take the agent across possible large distances in hidden state space to where it might not have been before. Hierarchical methods promise to be a good approach to scaling up to larger tasks.

2.7 Discussion

In this chapter, we gave a short introduction to the POMDP paradigm and RL-related POMDP methods. We have presented several algorithms that deal with model-free POMDP problems, and highlighted those that inspire our own algorithms in the remainder of this paper. Especially hierarchical approaches like HQ and SSS have our interest, since they promise to be 'the way to go' for future research.

Chapter 3

Utile Distinction Hidden Markov Models

3.1 Introduction

In this chapter we present and compare two novel algorithms, Utile Distinction Hidden Markov Models (UDHMMs) and Coupled Utile Distinction Hidden Markov Models (CUDHMMs). Both are based on the construction of anticipatory behavior models using a generalization of Hidden Markov Models (HMMs) (Rabiner, 1989). The nodes of these HMMs are used to model the internal state space (hidden states) and the transitions between them are used to represent the actions executed by the agent. Using these models, we can propagate *belief* about internal states during the execution of a trial. During a trial, we update the belief for every internal state at each time step. Building on this probabilistic framework, we modify the Baum-Welch parameter estimation procedure such that it enables the HMM to make *utility* distinctions, which amounts to a search for *relevant* discriminations between states. Superimposed on this model, we use a particular form of Reinforcement Learning (Kaelbling, Littman & Moore, 1996; Sutton & Barto, 1998) to be able to estimate the utility of actions for newly learnt hidden behavior states.

Previous work relevant to UDHMMs and CUDHMMs has been carried out by Chrisman (1992) and McCallum (1993, 1995a, 1995b). Chrisman first used a HMM to predict hidden world state. His Perceptual Distinctions Approach constructs a world model (HMM) that predicts observations based on actions, and can solve a number of POMDP problems. However, it fails to make distinctions based on utility — it cannot discriminate between different parts of a world that look the same but are different in the assignment of rewards. Chrisman posed the Utile Distinction Conjecture that claimed that state distinctions based only on utility would not be possible.

McCallum (1993) refuted that conjecture by developing a similar HMM-based algorithm that splits states based on their utility, Utile Distinction Memory (UDM). However, apart from being slow, UDM suffers from a severe limitation: while considering a state split, it only considers the previous belief state, while ignoring longer history traces. This renders certain POMDP problems which involve memory of more than one time step insoluble by UDM. McCal-

lum solved this problem by introducing USM (McCallum, 1995a) and U-tree (McCallum, 1995b). These algorithms construct variable-depth decision trees in which historical information is stored along the branches of the tree. The branches are split if a statistical test shows that splitting aids the algorithm in predicting future discounted reward (return or utility).

We present an approach that can make utility distinctions based on history traces of arbitrary length, while at the same time constructing a behavior model that can be used by any POMDP reinforcement learning method. The agent using UDHMM or CUDHMM is able to cope with noisy observations, actions, and rewards, and uses Baum-Welch to decide which features of the environment are relevant to its task. Furthermore, the algorithms are simple and perform well in the number of steps required for the agent to learn its task. The difference between UDHMM and CUDHMM lies in the type of Hidden Markov Model used. While for UDHMMs each node produces both observations and utility, CUDHMM uses a Coupled Hidden Markov Model (Brand, 1999) where two separate but coupled groups of states are used, one for observations and one for utility.

In section 2 we outline our utile distinctions approach and briefly describe the two algorithms. In section 3 we elaborate on the details of UDHMMs. Section 4 is reserved for explaining Coupled Utile Distinction Hidden Markov Models and discusses the differences between the two algorithms. Section 5 presents experimental results on two problems, where we show a very good performance of the UDHMM approach, especially in a highly stochastic domain. In section 6 we discuss the results and leave room for some speculation.

3.2 Utile Distinction and HMMs

The behavior models used by UDHMM and CUDHMM are both generalizations of the Input-Output Hidden Markov Model (IOHMM) (Bengio & Frasconi, 1995), an extension of the standard HMM, in which the agent's actions serve as input signals. The IOHMM in our case extends the standard HMM by allowing for transition probabilities between states conditioned by the actions. In this model, we represent actions as transition probabilities between states and we model observations for every node in the IOHMM.

So, using this IOHMM framework, how do we introduce the necessary utile distinction? It is important to allow for history traces of length more than one, in order to provide enough flexibility to cope with a complex environment. Simple environments may contain landmark observations that indicate utility to the agent. UDM is able to distinguish those. But if not one single landmark, but a whole chain of events would be indicative of expected utility, UDM would have considerable difficulty with the task, or might even fail to solve it. This asks for a solution.

But, not only do we want to be able to discriminate between longer history traces, we want to avoid unnecessary memory bookkeeping that slows down the performance, and might produce overfitting. We want memory to be created where needed, going back as long as needed.

For an elegant solution, we turn to the Baum-Welch procedure itself: by including the utility at every time step in the observation vector of every time step, we let Baum-Welch search for a model that predicts *both* observations and

utility, in relation to one another.

The general idea behind UDHMM and CUDHMM is as follows. We split up learning by constantly repeating two phases after one another, one for online learning (reinforcement learning during trials), one for offline learning (model learning between trials). When the agent is online, that is, acting in the world, the utility is not known yet. Only after the completion of a trial we can compute what the utilities were at every time step. So, during online performance, we simply *ignore* the utility factor. During online execution, the agent uses the forward-pass of Baum-Welch to update its belief over internal states. It observes the environment, chooses an action based on Q-values calculated by a version of SARSA(λ)-learning (Sutton & Barto, 1998), then updates its Q-values according to the perceived results and updated belief.

As soon as a trial is finished, offline learning — learning between trials — begins. It consists of updating the behavior model, the HMM, this time *including* the now known utilities. The utilities are handled as being part of the observation, and used in the re-estimation of the model parameters. Every internal state now not only models transition statistics for all actions and probability densities for observation vectors, but also probability densities for the perceived utility. Utility becomes part of the model.

Baum-Welch manages to create anticipatory behavior models that span multiple time steps. By including the utility in the observation, Baum-Welch is enabled to predict utility based on history, creating memory where needed. In order to be able to do this, the system needs spare states to be used for the creation of memory. By splitting states whose utility distributions are not Gaussian according to a statistical test, the algorithm creates its own state space to craft its utility predictions.

3.3 Algorithm Details: Utile Distinction Hidden Markov Models

The Utile Distinction Hidden Markov Model consists of a finite number of states $\mathcal{S} = \{s_1, s_2, \ldots, s_N\}$, a finite number of actions $\mathcal{A} = \{a_1, a_2, \ldots, a_K\}$ and a set of possible observations $\mathcal{O} = \mathbb{R}^{d+1}$. Note that with observations of dimension d, we add one element to the vector — the predicted utility — to obtain a vector of length $d + 1$. In case of discrete observations, as is the case in our experiments below, the observation vector will be a tuple $\langle n, r \rangle$ where n is a natural number indicating the observation and r a real number indicating the return. For every state s we keep transition probabilities $T(s_i, a_j, s_k)$ to other states for every action. For every state, there is an observation mean vector and a covariance matrix that together describe a Gaussian probability density function $p(o_i|s_j)$. There are also the mean utility $\mu_R(s)$ and the utility variance $\sigma_R(s)$, kept at every node s. The agent's belief at every timestep about hidden state is denoted by the belief vector $\vec{b}_t = \langle b_t(s_1), b_t(s_2), \ldots, b_t(s_N) \rangle$. See Figure 3.1 for a Bayesian representation of the model.

We compute the belief of the agent at every time step by using the forward-pass of the Baum-Welch procedure:

$$b_t(s_i) = \eta \cdot p(o_t|s_i) \sum_j T(s_j, a_{t-1}, s_i) b_{t-1}(s_j)$$

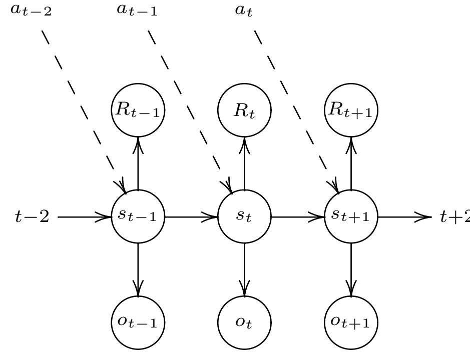


Figure 3.1: A dynamic Bayes net diagram representing graphically the relations between the different parts of a UDHMM. The influence of actions a_t is represented by dashed arrows.

where η is a normalization constant to ensure the belief values correctly add up to 1.

We use reinforcement learning for adjusting the agent's policy. Since we face non-deterministic environments in which state certainty is unavailable and the only information we have is a belief vector over internal states, we need to design a method to deal with this uncertainty. At every node s, for every action a, we store Q-values $q(s,a)$. Then, we define the Q-value of a particular action a for a particular belief vector $\vec{b}_t$ to be a result of a weighting process of all states:

$$Q(\vec{b}_t, a) = \sum_i b_t(s_i) q(s_i, a)$$

Using the belief vector, we let the agent update its Q-values by *linear* SARSA(λ)-learning, an approach similar to that of Loch and Singh (1998) but generalized to the linear case like in Littman, Cassandra and Kaelbling (1995). It uses eligibility traces $e_t(s,a)$ to update state-action values. On experiencing experience tuple $\langle \vec{b}_t, a_t, r_t, \vec{b}_{t+1}, a_{t+1} \rangle$ the following updates are made:

$$\forall (s, a \neq a_t): \quad e_t(s,a) := \gamma \lambda e_{t-1}(s,a)$$

$$\forall (s): \quad e_t(s,a_t) := \gamma \lambda e_{t-1}(s,a_t) + b_t(s)$$

$$\delta_t = r_t + \gamma Q(\vec{b}_{t+1}, a_{t+1}) - Q(\vec{b}_t, a_t)$$

$$\forall (s,a): \quad \triangle q(s,a) := \alpha e_t(s,a) \delta_t$$

where α is the learning rate and γ the discount factor.

So far the description of the online part of the learning algorithm. Now we consider the offline part, anticipatory model learning. We use Baum-Welch to update the model parameters such as the transition probabilities $T(s,a,s')$, initial occupation probabilities $\pi(s)$, and the observation probability density

function $p(o|s)$ parameters. But we ought to include the utility as well. We define an approximation to expected utility, future discounted reward or *return*, to be:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^n r_{t+n} + \ldots$$

These are the values we are going to model. For if the *distributions* of returns from one state are significantly different depending on what history preceded it, the state is not a consistent, Markovian state in that it may require a different policy depending on its history. In order to make the state Markovian again, UDM resorted to state splitting, while UDHMM leaves this to Baum-Welch (though UDHMM uses state splitting as well, as described below, in order to create enough hidden states for Baum-Welch to work on): we include the probability density function (pdf) of the return in the observation probability densities for Baum-Welch parameter re-estimation:

$$p(o_t, R_t|s) := p(o_t|s) \cdot \mathcal{N}(R_t, \mu_R(s), \sigma_R(s))^\theta$$

where $\mathcal{N}(x, \mu, \sigma)$ is a function for variable x in a Gaussian bell curve with mean μ and variance σ. θ is a parameter indicating the importance and impact of utility modeling relative to observation modeling — in our experiments, we found that when θ was too high, the perception modeling degraded too much, while when too low, utility prediction failed. By replacing the observation pdf with the combined observation-utility pdf, we enable the algorithm to use Baum-Welch to make utility distinctions that span multiple time steps.

It should be noted that, as the agent becomes more adept at its task, the utility statistics change for hidden states. This means that Baum-Welch should preferably be performed on the latest trials, since trials from its early history might distort utility modeling.

In order to be able to predict utility, the algorithm needs enough nodes in the HMM in order to create memory. This means that some kind of heuristic for state-splitting would be appropriate. We choose to split states that do not have a Gaussian return distribution. This means return statistics must be gathered in order to be able to do this test.

Determining whether and how a state should be split involves a statistical test to falsify beyond reasonable doubt that a state's utility distribution is not *normal* (i.e., Gaussian). If a split is performed, the EM algorithm is used to find the best fit to the utility distribution, and state splitting occurs according to the found mixture components. Transition probabilities are distributed evenly among the resulting split states.

We split states after the Chi-Square test on a discretized return distribution for a node shows that the return distribution is not a Gaussian. When this happens, the UDHMM invokes the EM algorithm on the node, where mixtures of Gaussians with various (2, 3 and 4) numbers of components are used to determine the best fit for the distribution. We define 'best fit' to be the Gaussian mixture with the smallest number of components that still offers an acceptable fit. After this is determined, the state is split, where a new state is created for each component. Transition probabilities are distributed equally over the newly created states. After each state-splitting, Baum-Welch is re-invoked to improve the overall model.

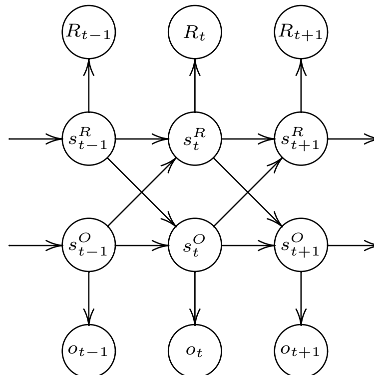


Figure 3.2: A dynamic Bayes net diagram representing graphically the relations between the different parts of a Coupled UDHMM. We have omitted the actions for readability. We can see that the internal state space is factorized in nodes $\mathcal{S}^O$ modeling observations and nodes $\mathcal{S}^R$ modeling utility.

This state-splitting leads to enough *excess states* to enable the algorithm to discriminate good policies. It also does lead to some splits that do not enhance performance. However, it is necessary to have excess states in order to be able to discover more complex relationships between events and utilities (compare excess states with fringe nodes in McCallum's U-tree). Also note that this, in our experience, rarely leads to more than twice the number of nodes that is strictly necessary for performing a task.

3.4 Algorithm Details: Coupled UDHMMs

The Coupled version of UDHMM uses the Coupled Hidden Markov Model by Brand (1997) as the basis of its model. A Coupled HMM (CHMM) can be used to model parallel streams of data in relation to one another. A heuristic is used in order to link the two streams of data and in order to couple the transition probabilities of the states, such that the forward-backward procedure and the Baum-Welch re-estimation of the model parameters for both groups of states get integrated. Look at Figure 3.2 for a DBN representation of our CHMM.

In our case, one group of states $\mathcal{S}^O$ is used to model the observations, while another group of states $\mathcal{S}^R$ is used to model the returns. One could see this as a simultaneous modeling of 'observation space' and 'utility space', where the couplings between the two spaces ensure that both spaces develop hidden states and internal structure that tend to be relevant to prediction or anticipation in the other space. By capturing the statistical relationships between the two spaces, the algorithm develops a notion of *relevance*. By concentrating its modeling efforts on those parts of the world that tend to affect return, the agent develops a more useful behavior model than it would otherwise have.

As with the non-coupled UDHMM, the algorithm has an online and an offline phase. When online and acting in the environment, the agent uses only the $\mathcal{S}^O$ part of the model to update its belief at every step. The $\mathcal{S}^O$ nodes are also the

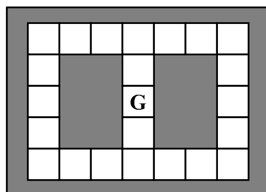


Figure 3.3: The Hallway navigation maze. The agent randomly starts each trial at one of the four corners. It observes its immediate surroundings and is capable of the actions goNorth, goEast, goSouth, and goWest. It must reach the center (labeled with 'G') to gain a reward.

nodes where the Q-values are stored and where Q-learning takes place. However, during the offline phase, that is, between trials, the $\mathcal{S}^R$ part of the model plays a role in the Baum-Welch re-estimation such that cross-modeling of the two spaces occurs and the $\mathcal{S}^O$ part of the model starts to be able to discriminate between situations with similar event characteristics but with wildly different return properties. Thus utile distinction is made by the algorithm.

State splitting is less easy to do for CUDHMM than for UDHMM, since two groups of states are involved. We chose to split $\mathcal{S}^O$ states the same way as we did for hidden states in UDHMM, except that for every split in $\mathcal{S}^O$, we created an extra node in $\mathcal{S}^R$, with randomized transition probabilities but with R-modeling initialized at the return statistics of the split state. After splitting, we let Baum-Welch re-estimation figure out how to fit the model including the new node.

3.5 Experimental Results

Utile Distinction Hidden Markov Models have been tested in several environments, of which three are described below. The first environment is a deterministic maze, which has previously been described and successfully solved by McCallum (1995a). The second is a maze especially designed to test how far back in time the algorithm can discriminate landmark observations. The third is a large and extremely stochastic navigation environment with 89 states, for which we show very good results with UDHMM.

3.5.1 Hallway Navigation

The first problem we tested was McCallum's hallway navigation task (McCallum 1995a). In this task (see Figure 3.3), the agent starts in one of the four corners of the maze. It can only detect whether there is a wall immediately north, east, south, and/or west of itself, so there are $2^4 = 16$ possible observations. It can perform four different actions: goNorth, goEast, goSouth, and goWest, which move the agent in the indicated direction. Its objective is to reach the goal (it gets a reward of 5 there), labeled with 'G' in the figure.

For a number of settings, we ran 21 experiments, which all consisted of many online trials and offline between-trial Baum-Welch steps. During each offline phase, we ran Baum-Welch only for the last 12 trials in order to save

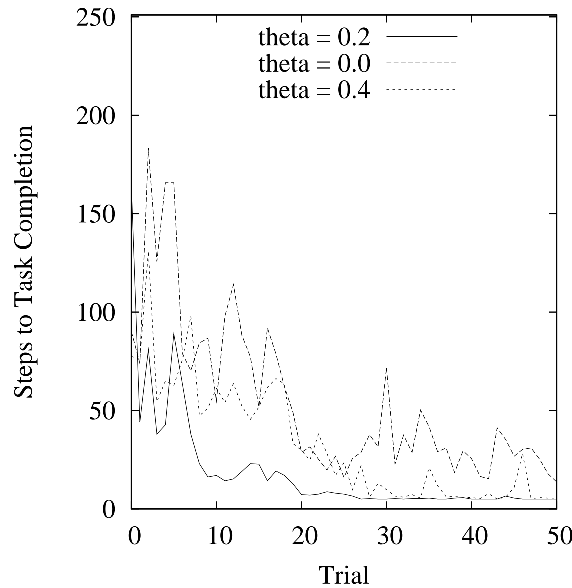


Figure 3.4: Performance of UDHMM on the Hallway Maze for different values of θ. We plot the number of steps it takes to reach the goal.

computation time and to prevent the model from settling down in a local sub-optimum (world dynamics change over time because the agent's policy changes as well). Of those 21 runs we plotted the one with median performance, where performance is defined as the number of trials it takes for the algorithm to consistently have an average number of steps to the goal under 7.0.

In Figure 3.4 we see the results for the UDHMM algorithm with different values of θ. When $\theta = 0.0$ we observed that in most runs, the algorithm does not reach the success criterion within 100 trials. This is because with $\theta = 0.0$, return is effectively ignored and cannot help with modeling utile distinction. Relying only on observations, the chances of finding a suitable model after all become slim. So clearly, utile distinction is needed to ensure effective behavior modeling.

When we tried $\theta = 0.4$, the algorithm did find a correct performance in all runs, but later than with $\theta = 0.2$. This is probably due to the fact that, certainly in the first few trials, a too large emphasis on return modeling causes too much modeling of noise in the return values.

The plot in Figure 3.5 shows the results of UDHMM with added noise. Noise consisted of a 0.1 probability of performing a random action, a 0.1 probability of observing a random observation and at every step a random number from the range -0.1 to +0.1 added to the reward. It also shows the results of CUDHMM on this problem. The median CUDHMM run is significantly slower to reach the success criterion than UDHMM, and in 7 out of 21 runs it did not reach the success criterion at all. We argue that this is because the S^R nodes tend to model R-values for too many $\mathcal{S}^O$ nodes and therefore lack discriminative power. Also we think that the node-splitting heuristic is much better for UDHMM than for CUDHMM. Utile distinction seems to function better with UDHMM.

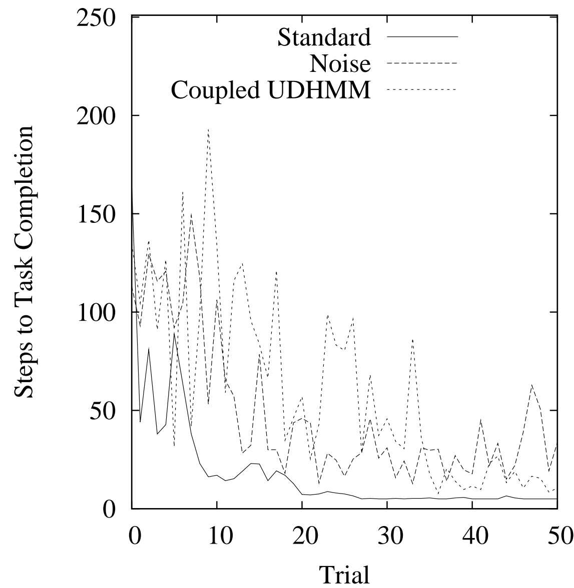


Figure 3.5: Results for standard UDHMM with $\theta = 0.2$ ('Standard'), results for UDHMM in a Hallway Maze with noise added to the observations, actions and rewards, and results for Coupled UDHMM in a Hallway Maze without noise.

When compared to the results of McCallum (1995a), we must concede that his USM achieves success in about 4 times as few trials. We do note, however, that our algorithm is ideally suited to noisy stochastic environments, which USM and Utree are not. The next problem will illustrate this.

3.5.2 Detecting Long-Term Dependencies: The T-Maze

The T-Maze task (see Figure 3.6) is taken from Bakker (2004), and is meant to test the ability to detect long-term dependencies. The task description is as follows. The agent starts at starting position 'S', there observing either the observation 'up' or the observation 'down'. This indicates whether the agent should go up or down at the end of the corridor, receiving a reward of 4 if it does so correctly and receiving punishment of -0.1 if it takes the wrong direction.

The initial observation is at least N (the length of the corridor) steps away from making the decision up-or-down, so this provides a useful test for checking the number of steps the algorithm can 'look back' in history. A history suffix algorithm like U-tree (McCallum, 1995b) certainly could not handle this since it would require a fringe depth of at least N, while algorithms like HQ-learning would not have much difficulty with it. Theoretically, HMMs (and therefore UDHMM) can look arbitrarily far back in history, but in pratice this ability is severely hindered by local suboptima in the HMM model and noisy perceptions.

We tested UDHMM (with $\theta = 0.2$) for several values of N. For every value of N, we ran the algorithm 21 times. The results are plot in Figure 3.7. The reasonable performance of UDHMM for the higher values of N cannot

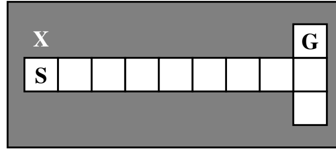


Figure 3.6: The T-Maze task. The agent observes its immediate surroundings and is capable of the actions goNorth, goEast, goSouth, and goWest. It starts in the position labeled 'S', there and only there observing either the signal 'up' or 'down', indicating whether it should go up or down at the end of the alley. It receives a reward if it goed in the right direction, and a punishment if not. In this example, the direction is 'up' and N, the length of the alley, is 8.

be entirely attributed to the Baum-Welch algorithm, for the reasons sketched above. Therefore, the explanation must be sought in the combination of Baum-Welch, state-splitting and utile distinction modeling. Still, for values of N above 10, the algorithm performs poorly, certainly in comparison with Bakker's (2004) RL-LSTM neural net method, that can handle values of N up to 50.

3.5.3 The 89-State Maze

This problem is taken directly from Littman, Cassandra and Kaelbling (1995), where seeded Linear Q-learning is applied on the task. It involves a maze in which the agent, next to position, also has an orientation, five actions Forward, TurnLeft, TurnRight, TurnAbout, and doNothing. It starts each trial at a random location with random orientation, and its goal is to reach the goal labeled 'G' (see also Figure 3.8). The actions and observations are extremely noisy.

Seeded Linear Q-learning's good performance can be explained by the avail-

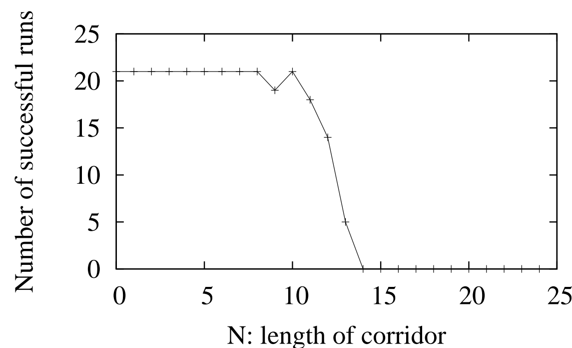


Figure 3.7: Results for the UDHMM on the T-Maze. For every value of N, the algorithm is run 21 times. Here we show the number of successful runs.

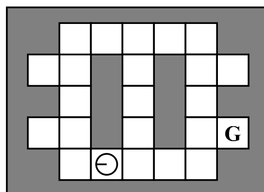


Figure 3.8: The 89-state maze. In this maze, the agent has a position, an orientation, and can execute five different actions: Forward, TurnLeft, TurnRight, TurnAbout and doNothing. The agent starts every trial in a random position. Its goal is to move to the square labeled 'G'. The actions and observations are extremely noisy, for example, there is only about 70% chance that the agent will get an observation right. For the precise world model we refer to elsewhere.

ability of the world model to the algorithm. Loch and Singh (1998) showed very good results (see Table 1) with a simple model-less and memory-less SARSA(λ) approach, where Q-values are stored not for states but directly for observations.

We took Loch and Singh's parameters (exploration method, λ, α, γ) and generalized the algorithm to our UDHMM version with linear SARSA(λ) by replacing direct observations with UDHMM's hidden states. We set $\theta = 0.2$, and let the state space expand from an initial 25 up to a maximum of 70 hidden nodes. Every trial was allowed up to 251 steps. We ran 21 runs. The results for the median run (median in the sense of median percentage success in all test trials) are shown in Figure 3.9 and 3.10.

Of course, this algorithm performs worse than recent algorithms that do have a world model. A good comparison must therefore be sought among model-free algorithms. Within that domain, the results are similar to RL-LSTM (Bakker, 2004), a recurrent neural network approach that uses gating to handle long-term time dependencies. RL-LSTM yields very good results, although it is slow to converge. Our algorithm is among the best model-free algorithms available.

Table 3.1: Overview of the performance of several algorithms on the 89-state maze. Shown is the percentage that reached the goal within 251 steps, and the median number of steps to task completion.

Algorithm	goal%	median steps
Random Walk	26	>251
Human	100	29
Linear Q (seeded)	84	33
Sarsa(λ)	77	73
RL-LSTM	94	61
UDHMM	92	62

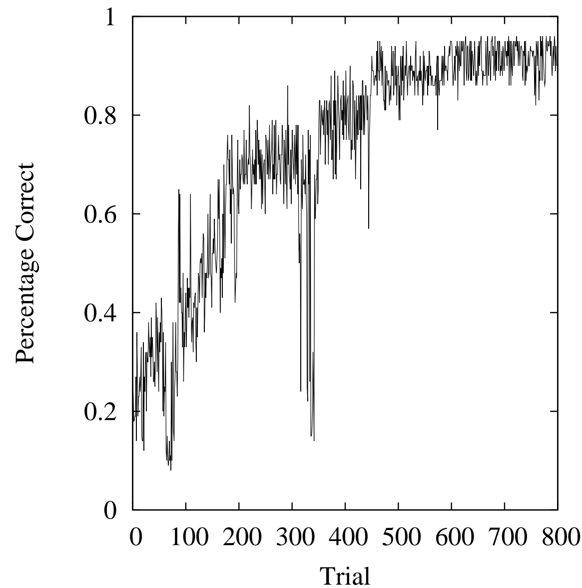


Figure 3.9: Results for the 89-state maze. Here we plot the percentage of tests that reach the goal within 251 steps.

3.6 Discussion

The computational complexity of UDHMM is quadratic in the number of states, per Baum step $\mathcal{O}(N^2T)$. This is not very bad but considering that many hundreds of Baum steps are needed even to solve small POMDPs, it is a rather slow algorithm. In this sense it compares unfavorably with, for example, U-tree, which is instance-based.

UDHMM's performance measured in the number of steps taken in the world in order to approach a good policy is very good for certain classes of problems, though it varies considerably due to possible suboptimal local maxima. Moreover, UDHMM, unlike U-tree, allows for a continuous multi-dimensional observation space in a very natural way and develops a probabilistic behavior model during its task performance. Utility modeling is done elegantly using Baum-Welch instead of using heuristic state-splitting methods. The UDHMM can also be used for other reinforcement learning methods such as other Q-learning variants or policy-gradient methods, which makes it a flexible tool for multiple POMDP approaches. Moreover, UDHMM includes a method for *selective attention* in multi-dimensional observation spaces: parts of the observation vector that are helpful to utility prediction tend to be modeled more precisely than parts of the vector that are not. Now UDHMM heavily biases the modeling of observations together with the modeling of utility, thereby leading to a kind of selective perception.

Much research has been directed at many different extensions to the basic HMM framework. Future work on UDHMM could include factorising the internal state space by using Factorial or Coupled Hidden Markov Models. We could think of hierarchical approaches, using Hierarchical Hidden Markov Mod-

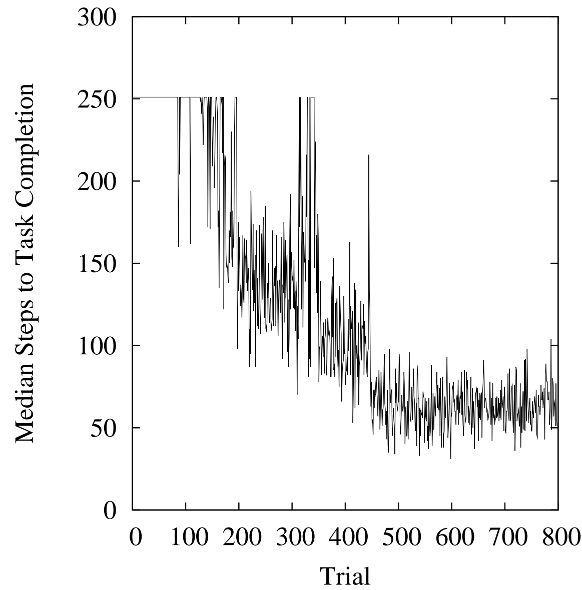


Figure 3.10: Results for the 89-state maze. Here we plot the median number of steps taken to task completion, i.e. reaching the goal. Note that the first trials have all 251 median steps associated with them, since if a trial takes longer than 251 steps, we break off and start a new trial.

els, that could increase detection of relevant long-term dependencies and could facilitate the re-use of (parts of) already learnt policies (see also Chapter 6 on Hierarchical UDHMMs). Several other EM methods seem applicable. Also, we could contemplate modeling utility in a different way, as a mixture of Gaussians, for example, or modeled per action. Or we could model *difference* in utility instead of utility per se. Considering all this, we conclude this is a very promising direction of research.

Chapter 4

Hierarchical Methods in Reinforcement Learning

4.1 Introduction

Many Reinforcement Learning methods suffer from the problem of not being able to easily scale up to larger problems. When environments are complex and tasks are difficult, the acting agent's exploration/exploitation trade-off becomes very difficult: multi-step exploration into unknown state space territory hardly ever yields immediate results — often very specific event sequences (state-action combinations) are required. And when there *are* positive reinforcement signals after a multi-step lucky guess, it is hard to attribute those to the right events. This makes the learning process very slow. Furthermore, when RL methods are applied to POMDPs, where no state certainty is given, only observations, then matters become even more difficult. In such environments, additional techniques are required to intelligently speed up learning.

One way to speed things up is to exploit a characteristic that is very common of both world structure and task structure: hierarchy. Hierarchy, repetition, and recursivity occur very often, so when one were to design a hierarchical approach for Reinforcement Learning, one could more easily explore those regions of state space that are otherwise hard to reach, namely those parts reached by intelligently repeating (parts of) already learned policies in different contexts, hoping that similarities in either world or task structure might deliver a lucky trial. Basically, hierarchical Reinforcement Learning is about assuming hierarchy in task or world structure, and then 'guessing' the right combinations of actions based on this hierarchical structure.

The advantages of hierarchical methods are to be found in the exploitation and reuse of (sub)policies. For example, when different parts of the world require similar action sequences, we could design a method for reusing policy building blocks, maybe adapted to changed circumstances. Or states could simply *share* parts of policies in a way that does not harm the optimal policy for either state. Or we could explore state space more effectively, by combining and recombining whole subpolicies never tried before.

Hierarchical algorithms also promise a method for naturally 'scaling up' to larger tasks by first learning simple things, then recombining those experiences

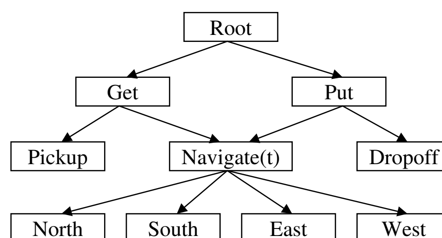


Figure 4.1: A MAXQ task hierarchy. In MAXQ, policies can share subpolicies of which the value functions are similar except by an offset. This enables the algorithm to reuse experiences or subpolicies for multiple tasks, and therefore speed up learning greatly. In this figure, both the 'Get' and 'Put' policies make use of the 'Navigate' policy (which can take an argument from ist parent policy). 'Navigate' only has primitive actions to select.

in order to achieve more complex goals. After all, a baby first learns to crawl, and only then to walk. Learning can more easily be seen as *incremental*, something which is essential to natural learning.

Because of all this, hierarchical Reinforcement Learning methods are usually better and faster in realistic environments. The problem is how to design this hierarchical algorithm. Preferably, we want the designer's role to be minimized, since minimizing human policy design costs is one of the main aims of Reinforcement Learning. So we want hierarchy to be created automatically, with as little human interference as possible. Unfortunately, this is a very hard, open problem. In this chapter we will present several approaches that attempt to do exactly this. But first we introduce some well-known algorithms that do need a fixed design for their hierarchical structures.

4.2 Hierarchical Methods: an Overview

One of the first frameworks that formalize the idea of hierarchical structure in Reinforcement Learning, is the hierarchical Options framework (Sutton, Precup & Singh, 1999). Sutton et al. use the notion of *temporal abstraction*, that is, they introduce actions that extend over multiple time steps, and adapt Q-learning rules accordingly. Imagine a tree-shaped policy tree, with the topmost 'agent' starting, choosing any of its children — options — as a temporally extended 'action'. This root agent does not select between normal actions anymore, as is common in plain Reinforcement Learning, but selects a child agent based on world state. Then that child can select among its own children yet another agent, and so forth, until the leaves, the primitive actions, are reached. An agent continues selecting its children or primitive actions until it decides, according to a given stochastic function β, to release control back to its parent controller. When the parent controller is given back control, it learns, with an extension of Q-learning that takes account of the number of time steps between initial agent activation and the release of control.

This framework can speed up learning enormously. However, the problem with this method is that Sutton et al. required the hierarchy and the release

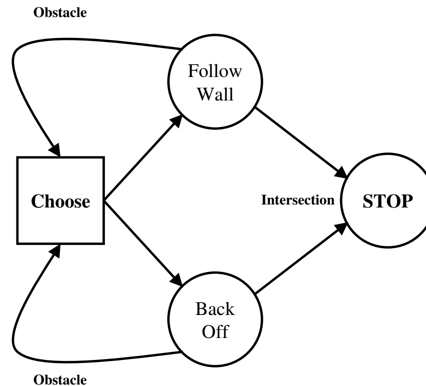


Figure 4.2: This figure shows the internal structure for a HAM agent. In HAM, there are many agents which can select ('call') one another. Agents can perform primitive actions according to the current internal state (for example, 'BackOff' or 'FollowWall' in this example), or release control ('Stop') to the agent that selected this agent, or select a subagent ('Choose').

function β to be designed by the programmer. Furthermore, *sharing* subpolicies for different parts of state space (for example, a 'leave room' action might be applicable in many rooms) is not very well possible with this method. This must be remedied.

Dietterich's (2000) MAXQ algorithm (see Figure 4.1) does provide a way of sharing substructure. MAXQ's agents, like in the Options framework, select subagents. But here the subagents can be shared *even though they are used in different parts of state space.* The key observation here is that, although value functions might differ depending on which parent activated the currently active agent, the *structure* of the value functions is very similar and differs only by a fixed offset. To take our 'leave room' example action again, the 'leave room' action could be activated by either the 'go to toilet' action or the 'go to work' action. The value function of 'leave room' differs depending on whether the agent wants to go to the toilet or wants to go to work, but, the structure is similar in that if one state's value is higher than another in one case (toilet), the same is true in the other case (work). The difference is the difference in expected discounted return after leaving the 'leave room' policy. This method Dietterich implements by using a *value completion* function to represent this difference.

Still, Dietterich's method has many elements that need to be designed by the task programmer, such as the hierarchical task structure and which subgoals each agent has.

Another interesting technique is the Hierarchies of Abstract Machines (HAM) (Parr & Russell, 1997) formalism. HAM consists of many agents, of which only one is active at any time. HAM allows each agent to maintain an 'internal state' (see Figure 4.2, with internal states 'FollowWall' and 'BackOff') that can be changed by abstract actions. The agent can perform normal actions, but can also invoke other agents ('Choose') or release control back to the agent that activated it ('Stop'). This rich language allows for procedure-like calls, even

with arguments, which makes the method very powerful in principle. However, also this method must be hand-designed to a great extent.

Hernandez and Mahadevan (2001) extended the HAM approach to a hierarchical nesting of several levels — Hierarchical Suffix Memory — where HAM agent selection at a certain level in the hierarchy takes place depending on the suffix of previous events experienced at that present level in the hierarchy. This is somewhat reminiscent of McCallum's Utree (1995b) where suffixes are also used as short-term memory. This use of suffixes, where event sequences at different time scales are used to discriminate between (hierarchical) states, enables the agent to deal with partially observable environments (POMDPs). Hernandez and Mahadevan (2001) show that this method works well on a mobile robot task. A lot a human design work is involved, though, in designing the hierarchical structure of the algorithm. However, it is suggested that any hierarchical Reinforcement Learning technique instead of HAM can be used, so methods for automatic hierarchy discovery might also be applicable.

The methods discussed so far, apart from requiring a lot of human design, also suffer from another problem: they do not deal well with stochastic environments. Very stochastic environments require the algorithm to be flexible as to where the control is, and to be able to change the hierarchical control easily when a few noisy observations led to the activation of the wrong policy. A wrong policy must then not be followed to the end, but stopped in time. The algorithms sketched above have difficulty with this. They rely on a clear, discrete control.

For stochastic partially observable environments, Theocharous (2002) uses a Hierarchical Hidden Markov Model (Fine, Singer & Tishby, 1998) to model a stochastic office environment at different scales, and introduces macro-actions to work on this environment. He shows that using this statistical approach at different levels of abstraction reduces entropy and therefore performance of the algorithm. The algorithm is especially good at mobile robotics tasks. However, actions are selected according to the most likely state, and the model requires a lot of hand-coding. This renders the algorithm less useful for truly model-free POMDPs.

4.3 Automatic Discovery of Hierarchical Structure

Often Reinforcement Learning tasks are too difficult for a human to program it out — the solution might simply not be known to him. Or it is the case that it costs too much time to hand-code and fine-tune hierarchical structure. In either case, we want hierarchical structure to be discovered by an *algorithm* instead of a human. And we want this to happen *automatically.*

Unfortunately, the automatic discovery of hierarchical structure in Reinforcement Learning is far from trivial. In the first place, deciding where and when to *end* a subpolicy must be decided — what constitutes a 'subgoal'? Second, what is the topology, the shape, of the hierarchy, and how is this augmented?

The first matter — what is a suitable moment to end a subpolicy — was first satisfactorily dealt with by HQ-learning (Wiering & Schmidhuber, 1997), which uses a chain of reactive agents successively activating one another. The

first agent takes control initially. Agents are reactive, in that their Q-tables are $Q(o, a)$ values for observation-action combinations instead of state-action combinations. Apart from standard Q-learning, agents in the chain 'learn' to end their own activations to select the next by a mechanism similar to Q-learning, but then using the values from the other agent's HQ-table. The learning rule uses a form of temporal abstraction similar to that used by Options (Sutton et al., 1999).

A chain of agents is somewhat restrictive. Sun and Sessions (2000) developed the SSS approach (Self-Segmentation of Sequences) that uses a truly hierarchical structure like the one used in Options, where agents can choose from several subagents. Every agent, apart from the regular Q-table, also has a CQ-table, which has two actions *continue* and *end* that decide whether to continue control at the current agent or to end control and return it to the parent. The CQ-values, like it is the case with HQ-learning, are learned in a similar manner to Q-learning, but taking values from the agent active at the next time step and applying temporal abstraction learning rules. The control-action (*continue* or *end*) with the largest $CQ(o, \cdot)$ value is selected.

Bakker & Schmidhuber's (2004) HASSLE algorithm offers a comparable algorithm with the difference that capabilities to reach certain subgoals are learned. High-level policies discover subgoals, low-level policies learn to specialize on reaching certain subgoals. This makes the method more flexible than SSS.

HQ, SSS, and HASSLE solve the problem of deciding where to release control, but neither approach has a method of augmenting hierarchical structure. They do not increase the number of agents nor do they change the relation between agents.

McGovern (2002) uses the notion of 'bottle neck states' to increase the number of subpolicies. Bottle neck states are states that occur often in successful (high-reward) trials but never in unsuccessful trials. Being observed only before good things tend to happen, these states might be indicative of a possible 'bottle neck' in state space in order to reach a great reward. For example, there are two rooms and between them a door. The agent is in one room, a reward in the other. During unsuccessful trials, the agent almost never goes through the door. During successful trials, the agent *must* have gone through that door. Therefore, statistically, we could find that door state to be a bottle neck state, since it tends to occur before rewards but not during unsuccessful trials.

McGovern's method assigns each found bottleneck state its own subpolicy, which enhances high-level exploration and leads to faster learning.

Digney (1996) designed Nested-Q learning. In Nested-Q learning, so-called 'skills' are developed incrementally that can be composed to form higher-level skills. It first starts with reactive policies, but incrementally, hierarchical skill selection takes over to form skill compositions. Methods like these offer the most general incremental learning structure that we could want.

The Hierarchical Model Operators (HMO) algorithm (Wierstra & Wiering, 2004, described later in this thesis) operates directly on the hierarchical structure of an HHMM-based (Fine, Singer & Tishby, 1998) behavior model. It splits states when they are found to be hierarchically 'inconsistent' with respect to future discounted reward. HMO incorporates several split operators that are used for different kinds of inconsistencies. This algorithm improves the behavior model incrementally by copying policy trees and then by adding more and more

distinctions between the old policies. This algorithm, like Theocharous' (2002), performs well with stochasticity and uncertainty, and scales well to larger tasks because of the incremental nature of HMO model development.

4.4 Discussion

Hierarchical structure is essential for scaling up to larger problem domains. When we want to tackle more difficult problems, we probably will have to rely on incremental learning techniques that are inherently hierarchical. However, we must not only focus on hierarchy, since not every problem can be described wholly in hierarchical (context-free) terms — there are many real-world cases where hierarchy could be used to represent the macro-structure of a policy, but where a sort of 'variable' outside the hierarchy is needed to take care of details. We think hybrid methods with more powerful state representations – with both hierarchical *and* non-hierarchical elements — are necessary to take the next step towards larger-scale problem domains.

We have presented several approaches to automatically discovering hierarchical structure. We hope these methods prove to be only the first of many new (future) algorithms directed towards this goal. Discovering structure should be one of the most important future topics in Reinforcement Learning.

Chapter 5

A New Implementation of Hierarchical Hidden Markov Models

5.1 Introduction

Hierarchical Hidden Markov Models as proposed by Fine, Singer and Tishby (1998) generalize the Hidden Markov Model (Rabiner, 1989) concept by introducing the possibility of hierarchical structure in the model. Hierarchical, recursive or repetitive structure occurs everywhere in the natural world, and it is therefore natural to assume that allowing HMMs to have hierarchical structure might help the modeling of sequences. Fine, Singer, and Tishby (1998) indeed show good performance on some problem domains for the HHMM. Hierarchical structure can reduce the number of necessary training samples enormously.

As for POMDPs, the main concern of this thesis, exploitation of hierarchical structure in the world could allow for scaling up to solving much larger POMDP tasks than previously possible with 'flat' methods.

The general structure of an HHMM is as follows. The HHMM has a hierarchical, tree-like structure (see Figure 5.1). In addition to the 'normal' production states $\mathcal{S} = \{s_1, s_2, \ldots\}$ already present in the HMM framework, a HHMM can also contain *internal* states $\mathcal{I} = \{i_0, i_1, \ldots\}$ and *end-states* $E = \{e_1, e_2, \ldots\}$. Only production states — the normal states — produce observations, while end-states and internal states provide the hierarchical structure that defines the relationships between the clusters of production states.

The HHMM can be viewed graphically as a tree-structure, where the root-node and all other nodes that have descendents in the tree are the internal states, while the leaf-states are either end-states or production states. Every internal state has one and only one child that is an end-state. The other children are either production states or other internal states. Imagine 'activity' to start at the root at time step 1, flowing down to the production states. Transition probabilities are defined for all internal states and production states: the probabilities of states activating *brother* states (i.e., states that have the same parent internal state). Activity flows from node to node according to the transition

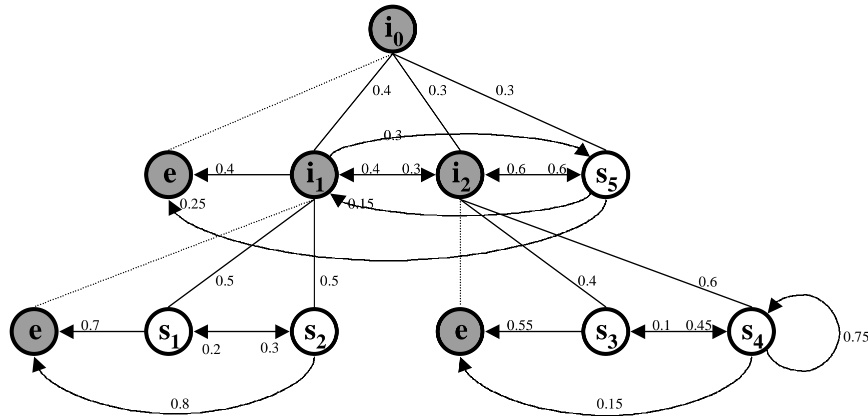


Figure 5.1: A three-level HHMM

probabilities. However, if a transition goes from a production or internal state s to end-state i, its activity is given back to its parent and passed on to uncles and/or grandparents, and so on. This way, every internal state recursively makes up a complete HHMM in itself.

In Figure 5.1 we can see a picture of a complete HHMM including transition probabilities. The horizontal arrows denote the horizontal transition probibilities between brother states, e.g. $T_{s_1 s_2}$ for the transition probability between s_1 and s_2. The connections between parent and child denote the probability that an internal node activates one of its children, e.g. $\pi_{i_2 s_3}$ denotes the probability that node i_2 activates s_3. All production states have observation probabilities associated with them, denoted $B(s, o)$.

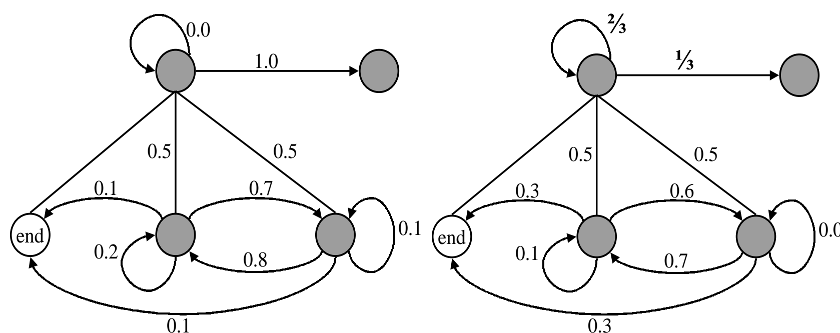
5.2 The Three Problems

As with standard HMMs, three problems are of interest concerning HHMMs:

- **Problem 1 :** Given the HHMM model $\mathcal{M} = \langle T, B, \pi \rangle$, how do we compute $Pr(O|M)$ for observation sequence O, where $O = o_1, o_2, \ldots, o_T$.
- **Problem 2 :** Given the HHMM model $\mathcal{M} = \langle T, B, \pi \rangle$, how do we choose a state sequence $S = s_1, s_2, \ldots, s_T$ so that $Pr(O, S|M)$ is maximized.
- **Problem 3 :** Given the HHMM model $\mathcal{M} = \langle T, B, \pi \rangle$, including hierarchical structure, how do we best estimate the parameters $\langle T, B, \pi \rangle$ so that $Pr(O|M)$ is optimized given experience sequences $O^{(k)}$.

The algorithmic solutions to those three problems given by Fine, Singer, and Tishby (1998) are far from perfect. In the first place, their generalized version of the Baum-Welch procedure (used for problems 1 and 3) takes $\mathcal{O}(NT^3)$ time which is very inefficient, and could become computationally infeasible for long sequences. Second, their algorithm is extremely complicated and very hard to implement.

In this chapter, we seek to develop a simpler, faster algorithm for re-estimating the model parameters.



This picture shows two equivalent Hierarchical Hidden Markov Models, of which the left one is minimally self-referential, while the right one is maximally self-referential. Maximally self-referential HHMMs can be easily converted to minimally self-referential HHMMs by appropriately redistributing transition probabilities, and vice versa.

Figure 5.2: Equivalent Hierarchical Hidden Markov Models

5.3 A new HHMM algorithm

The first step towards developing a better algorithm is noting the various possibilities for model equivalence. For example, figure 5.2 shows two equivalent HHMMs, that produce exactly the same observation sequence probabilities, and after conversion to a flat HMM, produce exactly the same flat models. The left tree is *minimally self-referential*, i.e. no internal node transitions to itself. The right tree is *maximally self-referential*, meaning the internal nodes' transition probabilities point maximally to themselves. Maximally self-referential HHMMs can be easily converted to minimally self-referential HHMMs by appropriately redistributing transition probabilities, and vice versa. We note that this possibility for multiple models that are essentially equivalent, is not necessary for a good model. We could, therefore, *reduce* the possibilities for model architectures without losing the model's expressive power.

So, the next step to improving the HHMM algorithm is to demand all HHMM models to be *minimally self-referential*: we do not allow self-referential internal nodes anymore. In that case, the *shortest* transition path between any two production states in the tree is at the same time the *only* path between them. When we now concentrate on production nodes, we can see that the *actual* transition probability between two production states can be computed from the shortest path through the tree. For example, the actual transition probability between states s_1 and s_3 in figure 5.1, denoted by $T^*_{s_1 s_3}$ (with asterisk), is in fact

$$T^*_{s_1 s_3} = T_{s_1 s_{end}} \cdot T_{i_1 i_2} \cdot \pi_{i_2 s_2} = 0.7 \cdot 0.3 \cdot 0.4 = 0.084.$$

This way we can easily construct all T^* between all production states. If we replace the usual T_{ij} with T^*_{ij}, we can, at every moment necessary, easily convert hierarchical model $\mathcal{M}^h$ to a flat HMM model $\mathcal{M}^f$, yielding composite model tuple $\langle \mathcal{M}^h, \mathcal{M}^f \rangle$. This provides an easy way out of generalizing problems 1 and 2 (sequence probability computation and best sequence with the Viterbi algorithm) by extracting from these values a normal HMM, to yield performance

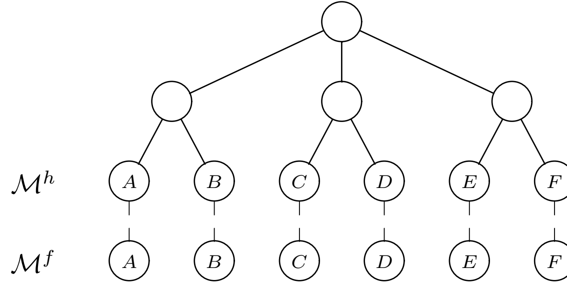


Figure 5.3: HHMM HMM couple $\langle \mathcal{M}^h, \mathcal{M}^f \rangle$ modeled in interaction. While converting the hierarchical model to a flat model, equivalent nodes still hold pointers to one another.

results similar to HMMs ($\mathcal{O}(N^2T)$) instead of $\mathcal{O}(NT^3)$. Using $\mathcal{M}^f$, derived from $\mathcal{M}^h$ to compute Problem 1 and Problem 2.

The third problem, hierarchical model-estimation, is more complex. For this problem we need to estimate transition probabilities not only between brother states, but also between far-off states, and between internal nodes, and from internal or production states to end-states. While for Problem 1 and Problem 2 we used $\mathcal{M}^h$ to compute $\mathcal{M}^f$, for hierarchical model re-estimation it is the other way around: we use $\mathcal{M}^f$ to compute flat values α (the forward-variable), β (the backward-variable), γ (state occupation probability), and ξ (state transition probability), which are then used to update the hierarchical nodes. This can be done, because for every flat node, there is exactly one corresponding hierarchical production state (see Figure 5.3).

In order to be able to calculate hierarchical transition statistics between two nodes in $\mathcal{M}^h$, we recursively add up transitions ξ between the flat nodes that are associated with all the *descendant* production states of those two $\mathcal{M}^h$ nodes. This way we can find transition statistics for every possible path between states in the tree.

We define observed state occupation probabilities γ_t (where $\gamma_t(i)$ denotes the probability that node i was active at time t) to be

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{Pr(O|M)}$$

and observed state transition probabilities ξ_t (where $\xi_t(i,j)$ denotes the probability that at time t, there was a transition from state i to state j) to be

$$\xi_t(i,j) = \frac{\alpha_t(i)T_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{Pr(O|M)}$$

for production states as for usual HMMs. Let $\sigma(i)$ denote the set of production state descendents of i in the HHMM, and let $\sigma(i) = i$ if i is a production node. A note on convenient notation: with $x \notin \sigma(y)$ we actually mean $x \in \mathcal{S} \setminus \sigma(y)$, i.e., all production states that are not descendents of y. Now we define $\gamma_t(i)$ for internal states to be the probability that, at time t, any of its descendents $\sigma(i)$ is activated by any state that is not a descendent of i. In order to do that, we sum over all possible transitions:

$$\gamma_t(i) = \sum_{k \notin \sigma(i)} \sum_{l \in \sigma(i)} \xi_t(k,l)$$

We can then hierarchically re-estimate the transition probabilities T_{ij} between all neighboring nodes i and j that are not end states. This involves hierarchically summing over all possible transitions between descendents of the involved states i and j:

$$\hat{T}_{ij} = \frac{\sum_{t=1}^{T-1} \sum_{k \in \sigma(i)} \sum_{l \in \sigma(j)} \xi_t(k,l)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

Estimating hierarchical transitions (compare with initial occupation probabilities π in flat HMMs) is done similarly, calculating the probability π_{ij} that transitions are made from anywhere, through i, to any descendent of j:

$$\hat{\pi}_{ij} = \frac{\sum_{k \in \sigma(j)} \gamma_1(k) + \sum_{t=1}^{T-1} \sum_{k \notin \sigma(i)} \sum_{l \in \sigma(j)} \xi_t(k,l)}{\sum_{k \in \sigma(i)} \gamma_1(k) + \sum_{t=1}^{T-1} \sum_{k \notin \sigma(i)} \sum_{l \in \sigma(i)} \xi_t(k,l)}$$

Let $p(i)$ denote the immediate parent of node i. Then, we re-estimate end-transitions by summing up over all transitions that do not lead to a child of the immediate parent:

$$\hat{T}_{i,end} = \frac{\sum_{t=1}^{T-1} \sum_{k \in \sigma(i)} \sum_{l \notin \sigma(p(i))} \xi_t(k,l)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

With these redefinitions and formulas we solve the re-estimation steps in linear ($\mathcal{O}(N^2T)$) time.

5.4 Discussion

Our implementation of HHMMs provides a simple and fast method for dealing with HHMMs. Murphy (2001) also developed a linear-time algorithm for re-estimating HHMMs. His method works in a totally different manner, and is based on dynamic Bayes nets. This is fine, but much harder to implement. That is why we chose to develop this algorithm.

A useful extension to the HHMM concept might be a layered Hidden Markov Model: a HHMM where some nodes might have multiple parents. In other words, where some internal states share substates. In this model, we have more advantages of *reuse*, as it speeds up learning and the finding of structure. Good approximations with this layered model could, because of forced substructure, possibly generalize better and therefore learn faster.

Chapter 6

Model Operators and Hierarchical UDHMMs

6.1 Introduction

In this chapter we describe a novel algorithm for solving model-free POMDPs hierarchically, by discovering hierarchical structure automatically via a set of heuristics for hierarchically splitting behaviour state space.

Model-free POMDP algorithms have so far been remarkably unsuccessful in scaling up to larger problem domains. Even the simplest tasks seem to elude the best POMDP algorithms when a model of the environment is unavailable. Policies that are found with flat methods usually get stuck in local maxima and perform poorly on a global scale. One could say that they fail to explore the more 'interesting' parts of hidden state space — the parts that rely on repetition, recursivity and hierarchies of policies. Re-use of parts of a policy is difficult in a flat framework. POMDPs are simply too complex to be solved straightforwardly.

What is needed is a set of extra assumptions on the problem domain, which should simplify the tasks to be solved. However ugly these assumptions may be theoretically, one assumption that might not be too restrictive, and applicable to most real-life problems, might be that of repetition and hierarchical structure. Repetition is found everywhere in nature, and repetitive structures in the world seem to offer methods of abstraction and generalization for any realistic agent. Hierarchical structures are also found everywhere, and provide the same evidence that a simplification of policies is possible. Even a bottom-up construction of a policy hierarchy might be found using underlying hierarchy in world conditions and task structure.

So hierarchical approaches seem to promise a method for solving POMDPs without a model. What a hierarchical algorithm could do is assume hierarchical structure in both task and environment and dramatically reduce the search space of the problem, thereby solving much larger problems than feasible with 'flat' structure algorithms. However, there have been few methods to discover hierarchy in POMDPs automatically. The issue of automatic discovery of hierarchical structure still remains an open problem. But there have been some advancements. POMDP algorithms that do address the hierarchy issue are, for

example, HQ-learning (Wiering & Schmidhuber, 1997), SSS (Sun & Sessions, 2000), and HSM (Hernandez & Mahadevan, 2001).

However, they all suffer from an inability to deal with stochasticity and with noisy unpredictable environments. Also, HQ and SSS find hierarchical structure but they are not very flexible in adapting their structure to changing circumstances or to incremental learning (i.e. bottom-up learning, where simpler tasks are learnt first in order to be able to solve larger problems). They lack a method for dynamic hierarchy creation.

In this chapter we present an algorithm that is ideally suited to stochastic environments with hierarchical structure. It keeps a hierarchical behavior model and implements a hierarchical learning method. Furthermore, it automatically updates its hierarchical *memory* model with Hierarchical Model Operators (HMOs) by *splitting* parts of its internal memory state space that are perceived to be *inconsistent*. Not only by splitting single states, but also by *hierarchically* splitting whole branches in memory state space. We designed a set of operator rules that enable the algorithm to learn faster and better in stochastic environments that exhibit repetition and structure. The splitting rules let the algorithm automatically (without any hand-coding) exploit the regularities the agent finds on its way, and makes the acting agent perform better in a number of cases. The algorithm also enables incremental learning and inductive transfer: it can re-use parts of its learned policies in order to find new policies. Here the underlying assumption is: trying something similar we already knew first is better than starting from scratch all the time.

The splitting rules are performed after rigorous statistical tests show that splitting might be useful to anticipating observations and *utility*. If there is strong evidence that a state split can provide the algorithm with a (possibly) better model on what effects (either on observations or on utilities) an action or event in the world might have, it splits.

Our method uses as its model a generalised Hierarchical Hidden Markov Model (HHMM) (Fine, Singer & Tishby, 1998), which we adapted to linear-time inference (see Chapter 4). We generalize that model with actions (IOHMM) and utility modeling like in Chapter 3. As is the case with UDHMM, it is an anticipatory model, a behavior model, not a precise model of the environment. However, this time the model is hierarchical and split operator rules are added. We show this to be a huge benefit to several POMDP tasks.

To outline the rest of the chapter, we start with a discussion on the hierarchical model used — a Hierarchical UDHMM (HUDHMM). Its structure, the way it is learnt, the learning rules that are used, those are all very similar to UDHMM's, except that it is generalized to HHMMs. After that, we describe the HMO-algorithm, which encompasses the rules for several Hierarchical Model Operators that are designed to enhance the model. We present experimental results for two domains. We end the chapter with an analysis of the performance of the algorithm, and discuss its present limitations and possible future extensions. We end with the conclusion that hierarchy-operators (HMO) provide a viable method for policy-improvement in hierarchical cases.

6.2 A Hierarchical Behavior Model: HUDHMM

Just like it was the case with UDHMM, we use a behavior model instead of a world model. That means that we model an internal memory state space that is not necessarily as complex as the world. It only suits as a model for what to do given the circumstances. It links observations, rewards, and actions, together in a model that can anticipate not only the consequences of events, but can also guess as to the usefulness of being in certain internal states.

As the basis of our model we use an extended version of the Hierarchical Hidden Markov Model (Fine, Singer & Tishby, 1998). Hierarchical Hidden Markov Models as proposed by Fine, Singer & Tishby (1998) generalize the Hidden Markov Model concept by introducing the possibility of hierarchical structure. Hierarchical structure allows for scaling up to larger task domains. We used our own version of HHMM algorithm that works in linear time (see Chapter 5) instead of cubic (!) time.

We extended this HHMM by allowing transitions to be conditioned by actions, much like we do in UDHMM. This is yet again an implementation of Input-Output Hidden Markov Models (Bengio & Frasconi, 1995) where actions take the form of signals influencing the state transitions at every time step.

We do not consider hierarchical nodes or end-states to be true memory states. The memory states that the agent uses to determine its situation are made up of all production states at the leaves. We can define a belief vector $\vec{b}_t = \langle b_t(s_1), b_t(s_2), \ldots, b_t(s_N) \rangle$ over N memory states $\mathcal{S} = \{s_1, s_2, \ldots, s_N\}$. The model includes a finite set of actions $\mathcal{A} = \{a_1, a_2, \ldots, a_K\}$ and a set of possible observations $\mathcal{O} = \mathbb{R}^{d+1}$. At every memory state s_n we keep statistics for utility, and we extend the observation vector o_t at every time step with one element r_t to mean the perceived utility. So it is that we get a $d+1$ observation vector where observation dimensionality is only d. We model and use utilities much the same way as we did for Utile Distinction Hidden Markov Models (see Chapter 3).

We compute belief at every time step by using the forward-pass of the hierarchical version of the Baum-Welch algorithm. With this belief vector, we can implement Reinforcement Learning (Sutton & Barto, 1998) for the agent in order to obtain, through continuous trial-and-error, a good and ever better policy.

We are dealing with stochastic and noisy worlds, and our memory belief state and the memory state model are imperfect. The belief vector does not tell us all of the environment that might be relevant. We need a Reinforcement Learning method that copes with that.

We use *linear* SARSA(λ)-learning, just like UDHMM. At every node s, for every possible action a, we store Q-values $q(s, a)$. Then, we define the Q-value of a particular action a for a particular belief vector $\vec{b}_t$ to be a result of a weighting process of all memory states:

$$Q(\vec{b}_t, a) = \sum_i b_t(s_i) q(s_i, a)$$

Using the belief vector, we let the agent update its Q-values by *linear* SARSA(λ)-learning. It uses eligibility traces $e_t(s, a)$ to update state-action values. On experiencing experience tuple $\langle \bar{b}_t, a_t, r_t, \bar{b}_{t+1}, a_{t+1} \rangle$ the following updates are made:

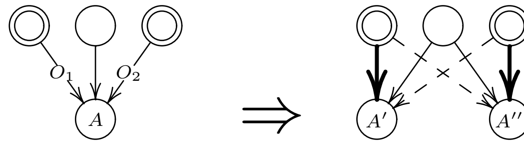


Figure 6.1: The Observation Split Model Operator. This operator is invoked in order to improve the model's predictive power concerning perceptions. It is triggered when a state is clearly not 'consistent' with regard the previous active node: when observations are different conditioned by the condition states, then it must be split. Legend with this illustration: Condition states are shown with a double circle. The state in consideration of splitting is node A.

$$\forall(s, a \neq a_t) : e_t(s, a) := \gamma\lambda e_{t-1}(s, a)$$

$$\forall(s) : e_t(s, a_t) := \gamma\lambda e_{t-1}(s, a_t) + b_t(s)$$

$$\delta_t = r_t + \gamma Q(\vec{b}_{t+1}, \vec{a}_{t+1}) - Q(\vec{b}_t, \vec{a}_t)$$

$$\forall(s, a) : \triangle q(s, a) = \alpha e_t(s, a)\delta_t$$

where α is the learning rate and γ the discount factor.

So far we have hardly diverged from Utile Distinction Hidden Markov Models except that we use a *hierarchical* HMM now instead of a flat Hidden Markov Model. The online and offline phases of the algorithm are similar, SARSA(λ)-learning is the same. But it is not the similarities that are important here. We could use any hierarchical POMDP algorithm to illustrate the idea of HMOs. The real difference lies with the application of Hierarchical Model Operators.

6.3 The Hierarchical Model Operators (HMO) Algorithm

In this section we describe the different Hierarchical Model Operators HMO that shape up the hierarchical model. Every operator changes the model in such a way as to improve its assessments of the world from the agent's perspective, or at least not deteriorate it. We design the rules such that the operators hardly ever decrease model quality.

6.3.1 The Observation Split Operator

The Observation Split operator is the simplest operator. It splits a state when, given two different previous state activations, significantly different observation distributions occur. In other words, if a state's observations do not seem to be independent of which state preceded it (what we call the two *condition states*), it is inconsistent and must be split. (See also Figure 6.1). With 'significant'

we mean a statistical test, the Chi-Square test on observation probabilities. In order to be able to do this, the algorithm must keep conditional observation statistics during execution.

When a split is performed, the state is split into two. New observation parameters for each of the nodes are determined as they would have been given the activation of one of the preceding states. Outgoing transition probabilities from each new state are copied from the ancestor state. Incoming transitions are distributed evenly, except for transitions from the two transition states. Both *condition states* create two connections, to every newborn (split) node. For each condition node, one of those new outgoing connections gets 95% of the strength of the original connection strength, that is, the node that 'fits' the observation distribution best, the other one gets 5% of the strength. Note that this is a heuristic for improving the agent's observation model. After the heuristic split is applied, Baum-Welch will further optimize model quality.

The Observation Split Operator allows the model to grow large enough to represent sufficiently complex event sequences for simple domains. For anticipating utile distinction it is not suitable.

The Observation Split is a near-absolute necessity for the HMO-algorithm. Without it (omitting Observation Split from the list of available HMOs), our experiments did much worse. This can easily be explained, since in a tree-like model, every node has a limited number of immediate brother nodes, and while hierarchical transitions can be used, it is good to have brother states which can together model coherent, logical subsequences.

6.3.2 The Brother Split Operator

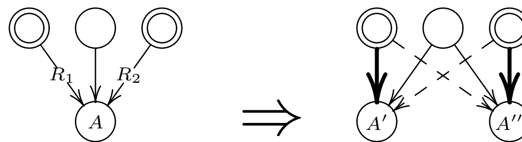


Figure 6.2: The Brother Split Model Operator. This operator is invoked in order to improve the model's predictive power concerning utility. It is triggered when a state is clearly not 'consistent' with regard to the previous active node: when return is different conditioned by the previously active condition states for some outgoing action, then it is judged not to be Markov and must therefore be split. Legend with this illustration: Condition states are shown with a double circle. The state in consideration of splitting is node A.

First devised by McCallum (1993) in UDM (Utile Distinction Memory), this method splits states if it helps predicting utility. It is based on the principle that internal states should be Markovian, that is, a state alone should be sufficient to correctly describe the agent's situation. And the situation defines a state's utility distribution. So, if a utility distribution for outgoing actions is different given condition states, then the state is not Markovian and must be split. See also Figure 6.2. Note that in order to be able to make this check, the algorithm must keep conditional utility statistics during execution.

We test the utility distributions against one another by using the Chi-Squared test on a discretized representation of utility distributions per incoming *condition node* and per outgoing action. If the test shows the state, given condition states, is not Markovian, it triggers this HMO and the state will be split. The resulting newborn states' incoming transitions from condition states are divided according to similarity (according to the Chi-Square test) in expected utility distribution. Other transitions are distributed evenly. After the operation, Baum-Welch is, again, reapplied in order to optimize the model.

The Brother Split greatly enhances the performance of our algorithm. Without it, results are very poor, because not enough new (utility-distinctive) states are generated during execution of the algorithm. Including Brother Split in our list of available HMOs is therefore a must.

6.3.3 The Uncle Split Operator

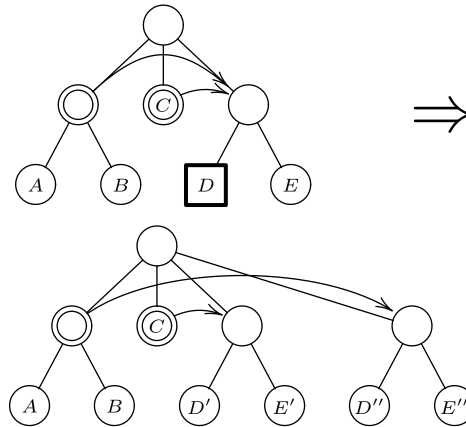


Figure 6.3: The Uncle Split Model Operator. When a hierarchical utility consistency check judges a state not to be Markov with respect to return, its entire branch will be split. Legend with this illustration: Condition states are shown with a double circle. The state in consideration of splitting is the square node.

This is a truly hierarchical split, that does not split a single state alone but a whole branch of hidden state space. It works as follows. If, for some outgoing action from the state in consideration (in Figure 6.3 shown as a square), the distribution of utility is different depending on which 'uncle' node was active one time step ago, the entire branch of which the state is a part must be split — see Figure 6.3. Transition strengths are divided appropriately, like with the Brother Split Operator, after which Baum-Welch re-estimation is invoked.

This operator gives the HMO algorithm the ability to re-use parts of the model (it copies an entire branch), which gives the algorithm the opportunity to use parts of hidden state space as building blocks for developing more complex policies. Explorations in state space look then more like reasoning over macro-actions (jumping to an entire new part of the tree). Furthermore, this increases the likelihood that long-term dependencies will be spotted.

The Uncle Split Operator is the essential ingredient of our hierarchical model

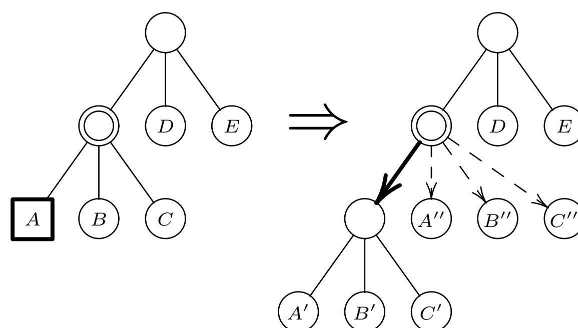


Figure 6.4: The Parent Split Model Operator. This is the only provided HMO that has the ability to create an extra hierarchy level. It is triggered when nodes are inconsistent with respect to utility depending on whether their activating parent was activated by another node or recursively by itself — assuming the parent is maximally self-referential. Legend with this illustration: The parent is shown with a double circle. The state in consideration of splitting is the square node.

building. Without it, the algorithm gets stuck in a local maximum of a relatively poorly anticipating model.

6.3.4 The Parent Split Operator

The Parent Split Model Operator is the only provided HMO that has the ability to create an extra hierarchy level. It is triggered when nodes are inconsistent with respect to utility depending on whether their activating parent p was activated by another node or recursively by itself — assuming the parent is maximally self-referential. If the operator is triggered, the children of p are copied, bundled into a new hierarchical node n and added as a child to p (see Figure 6.4). Vertical transition probability from parent p to new hierarchical child n are made very big, other vertical transitions from p are made very small. This way, a new level is constructed within the model hierarchy, while retaining model equivalence.

When provided initially with a diverse tree of several levels of hierarchy, Parent Split does not seem to have any big advantage on the results of our experiments. However, starting out from a smaller, more sober tree, the Parent Split Operator does lead to better performance.

6.3.5 The Cousin Split Operator

When an inconsistency is detected for *cousin* nodes, an operation within the hierarchical framework will not suffice for suppressing that inconsistency while at the same time retaining near-equivalence of the model before and after the operation. We need an extra memory placeholder outside the hierarchy to deal with this case.

Especially for Cousin Splits, we *factorize* our hidden state space by coupling (Coupled Hidden Markov Models (Brand, 1997)) our Hierarchical HMM with an extra flat HMM (see Figure 6.5), where every node in the flat model corresponds

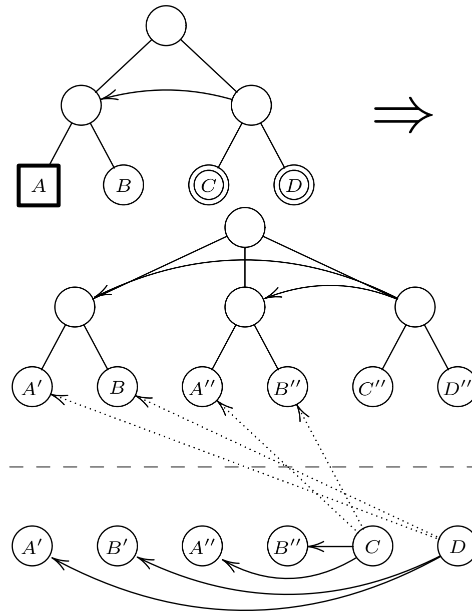


Figure 6.5: The Cousin Split Operator. When a node is deemed to be inconsistent with respect to its *cousin* condition nodes, it is split twofold: in the hierarchical model (above the dashed line) and in the coupled flat model (under the dashed line). Transitions between the two models are set appropriately. Factorizing memory space in a flat and a hierchical model enables us to deal with the cousin condition.

with a production state in the HHMM. Because of this flat part of the coupling, the algorithm can 'remember' which cousin it was that was active previously, and condition transitions accordingly.

The Cousin Split is in effect much like the Uncle Split, except for the special attention paid to the flat model, the transition probabilities of which are modified such as to 'remember' the cousin condition nodes that caused the split in the first place.

This illustrates the advantage of thinking not exclusively in terms of hierarchies, but also in terms of 'normal' memory (the flat HMM introduced here).

Since Cousin Splits need a coupling, we hold a Coupled HMM all the time. When other operations are performed, they are both performed on the hierarchical nodes *and* on the corresponding flat nodes.

Unfortunately, we have not experienced any different results for either the coupling or the Cousin Split Operation. Their effects seem negligible. However, since it is theoretically appealing, we keep this operator online and in the HMO set.

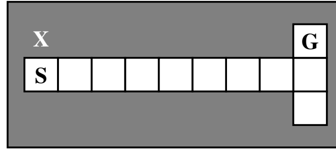


Figure 6.6: The T-Maze task. The agent observes its immediate surroundings and is capable of the actions goNorth, goEast, goSouth, and goWest. It starts in the position labeled 'S', there and only there observing either the signal 'up' or 'down', indicating whether it should go up or down at the end of the alley. It receives a reward if it goes in the right direction, and a punishment if not. In this example, the direction is 'up' and N, the length of the alley, is 8.

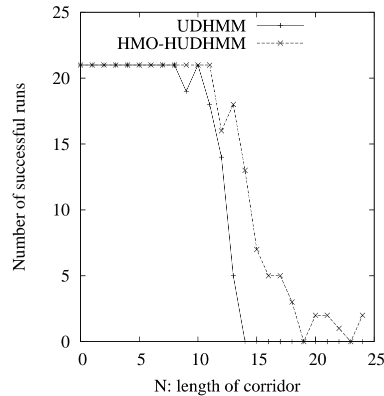


Figure 6.7: Results for the HMO-HUDHMM on the T-Maze. For every value of N, the algorithm is run 21 times. Here we show the number of successful runs.

6.4 Experimental Results

6.4.1 Detecting Long-Term Dependencies: The T-Maze

The HMO-HUDHMM algorithm was applied on the T-Maze task with corridor length N (see Figure 6.6) to check how far this algorithm could 'look back' in history as compared with other algorithms (see also Chapter 3). The task description is as follows. The agent starts at starting position 'S', there sensing either the observation 'up' or the observation 'down'. This indicates whether the agent should go up or down at the end of the corridor, receiving a reward of 4 if it does so correctly and receiving punishment of -0.1 if it takes the wrong direction.

For corridors of various lengths N, we ran 21 experiments. We used a random initial tree model with 20 production states with up to three levels of depth,

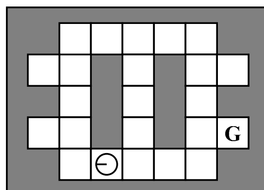


Figure 6.8: The 89-state maze. In this maze, the agent has a position, an orientation, and can execute five different actions: Forward, TurnLeft, TurnRight, TurnAbout, and doNothing. The agent starts every trial in a random position. Its goal is to move to the square labeled 'G'. The actions and observations are extremely noisy, for example, there is only about 70% chance that the agent will get an observation right. For the precise world model we refer to elsewhere.

and let that model grow up to a maximum of 60 nodes. For $\theta = 0.2$, we plot the results of 21 runs in Figure 6.7. We can see that this algorithm performs better than flat UDHMM. This is because 'lucky' splits by the HMO procedure often lead to the right hierarchical models that can more easily capture the long-term time dependency that is necessary for solving the T-Maze.

6.4.2 The 89-State Maze

The HMO-HUDHMM algorithm was also applied on the 89-State Maze (see Figure 6.8). See also Chapter 3 for details on this problem. For $\theta = 0.2$, and all other parameters inherited from UDHMM for this problem, we started out with a small random tree with 20 production states with up to three levels of depth. We allowed the model to gradually grow up to 150 nodes. To save computation time, we only used the last 50 trials in re-estimation of the hierarchical model parameters of the algorithm. We performed 21 experiments, below highlighting the one with median performance of those 21 runs.

Under these conditions, HMO-HUDHMM, with median steps 59, performed clearly somewhat better than flat UDHMM. For a comparison with other algorithms, see Table 6.1. However, since the 89-State Maze is very symmetrical yet distinctly different in utility in mirrored parts of state space, one could have hoped for a better result, certainly through the usage of the Uncle Split Operator.

Surprisingly, with $\theta = 0.0$ (no utility modeling at the nodes), the results are still good. We can conclude from this that hierarchical operators (HMOs) are a good match for utile distinction modeling as applied in UDHMM.

6.5 Discussion

HMO is a technique not only applicable on HUDHMMs, but probably also on other hierarchical RL algorithms, such as Dietterich's (2000) MAXQ value decomposition function, or Sun and Sessions' (1999) SSS algorithm. HMO could be a method for agent splitting or hierarchical policy splitting. Furthermore, it might be used on simple HHMMs: as we have experienced, the three operators Brother Split, Observation Split and Uncle Split, have greatly helped to improve

our HUDHMM model — the same might be true for a HHMM used in tasks like speech recognition.

HMO-HUDHMM performs relatively well on the problem domains investigated in this paper. It utilizes, by using HMO, a special technique for exploring behavior space: all of a sudden, in one split operation, an entirely new region of behavior space might become available to be exploited hierarchically. HMO-HUDHMM truly exploits hierarchy in world structure and in task structure, and it does so by forming hierarchies *automatically*, not hand-specified by a human programmer. It reminds of methods like SSS (Sun & Sessions, 1999) or Options (Sutton, Precup & Singh, 1999) where hierarchical policies enable hierarchical reasoning about macro-actions and policy combinations never tried before. HMO-HUDHMM is a bit like that, too, through its splitting procedures.

However, the hidden state space under HMO tends to get very big very quickly, and very inefficiently. For example, in our experiments, many nodes were often hardly ever visited. Possibly, by *sharing* HHMM substructures among several parent nodes, we could not only re-use parts of the model by splitting and copying, but by directly sharing. This could avert the algorithm from letting the number of hidden states get so big.

Another problem with HMO-HUDHMM is that fine-tuning of algorithm parameters takes a lot of human operator time.

Still, the algorithm is able to solve, for example, the 89-State Maze very well compared to other model-free POMDP approaches. We must conclude that investigations in hierarchical approaches yield fascinating opportunities for further improvements.

Table 6.1: Overview of the performance of several algorithms on the 89-state maze. Shown is the percentage that reached the goal within 251 steps, and the median number of steps to task completion.

Algorithm	goal%	median steps
Random Walk	26	>251
Human	100	29
Linear Q (seeded)	84	33
Sarsa(λ)	77	73
RL-LSTM	94	61
UDHMM	92	62
HMO-UDHMM (θ = 0.2)	93	59
HMO-UDHMM (θ = 0.0)	93	60

Chapter 7

Conclusion

Memory-based POMDPs offer a very general learning framework, for which the promise of ever better solution techniques, alas, remains elusive. In this paper we described three novel approaches to POMDP problems that seek to offer the ability to scale up to larger tasks, and supported this with the development of a new hierarchical HMM algorithm.

We introduced the concept of Utile Distinction Hidden Markov Models (UDHMMs), which offers a short-term memory capacity construction method for stochastic domains by discriminating utility over longer event sequences. This was done by using a Hidden Markov Model, as *memory*, to model not only observations and actions of an agent, but also to model the expected utility distribution for every hidden memory state. We show that this algorithm has good results for several domains, most notably stochastic ones, and succeeds in discriminating longer event sequences by their utility.

A related algorithm, the Coupled Utile Distinction Hidden Markov Model, seeks to do the same, but uses a two-stranded Coupled Hidden Markov Model instead, one strand for modeling 'world' events (observations and actions), and one for modeling utility. The philosophy behind this is that by separating the realms of events and utility, we can 'reuse' parts of event modeling space by 'coupling' it with the proper — separate — utility modeling. It could lead to an exploitation of the combination of event-symmetry but utility-asymmetry in certain tasks, such as the presented 89-State Maze. This algorithm performs significantly worse than UDHMM, however, contrary to our expectations. Our analysis shows this is because couplings between the two strands couple too many nodes that should have only weak impact on one another. Research on CHMMs by Zhong and Ghosh (2001) — who use a more sophisticated coupling computation scheme — suggests it might be possible to remedy this, something to which better tuning of the couplings might also contribute.

Scaling up memory-based POMDPs is still not satisfactorily solved. Possible solutions are most likely to come from hierarchical approaches, and we therefore turned our attention to hierarchical modeling techniques. Where UDHMM used the Hidden Markov Model as its basis, a hierarchical variant (HHMMs) was initially proposed by Fine, Singer and Tishby (1998). This algorithm has two flaws, one being extremely slow (cubic time), the other being very complicated and hard to implement. We developed and presented an algorithm that is both linear in time and easy to implement.

Using this HHMM framework, we generalize UDHMMs to the hierarchical case. In order to be able to expand hierarchical memory through experience, we introduced the Hierarchical Model Operators framework, which offers a method for augmenting the memory model topology appropriately 'on the fly', according to the agent's experiences. This enables the agent to be able to both discern longer-time dependencies, and to reuse parts of hierarchical memory (and therefore policies) in order to 'scale up' to more complex tasks. Our experimental results show this is indeed the case. The algorithm's experimental results on several tasks are very good.

Future Work

Our hierarchical algorithms perform very well compared to other memory-based approaches, but much work is still needed in order to boost performance. Also, parameter tuning is perceived as still being too cumbersome for our algorithms.

Sharing policies and sharing substructure might offer a way to reduce the time to find a (near-)optimal solution, and might prevent the memory model from growing too large too quickly. Such an approach would be better suited to exploit not only hierarchical but also repetitive 'world' and task structures.

But since world structures and task structures are usually not completely hierarchically describable, we still might need additional 'flat' memory, as our Cousin Split Operator shows. So far we have not found any evidence that 'factorizing' state space might be advantageous, but we are convinced that a proper factorizing of memory would be a major improvement for future POMDP algorithms. We suggest a hybrid approach of factorizing memory in both flat and hierarchical interdependent elements, possibly even including relational structures.

Memory-based POMDPs remain one of the most fascinating parts of AI research. They hold the promise of scaling up to much more complex learning tasks, and much interesting research will be done in the near future to achieve its goals.

Bibliography

Aberdeen, D. & Baxter, J. (2002). *Internal-state policy-gradient algorithms for infinite-horizon POMDPS* (Technical Report). RSISE, Australian National University.

Bakker, B. (2004). *The State of Mind*. Doctoral dissertation, Unit of Cognitive Psychology, Leiden University.

Bakker, B., & Schmidhuber, J. (2004). Hierarchical Reinforcement Learning Based on Subgoal Discovery and Subpolicy Specialization. *Proceedings of the 8-th Conference on Intelligent Autonomous Systems, IAS-8*. Amsterdam, The Netherlands

Barto, A.G., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete event systems, to appear*

Bellman, R. (1961). *Adaptive Control Processes.* Princeton University Press.

Bengio, Y., & Frasconi, P. (1995). An input/output HMM architecture. In G. Tesauro & D. Touretzky & T. Leen (Ed.), *Advances in Neural Information Processing Systems 7* (pp. 427–434). Cambridge, MA: MIT Press.

Brand, M., Oliver, N., & Pentland, A. (1997). Coupled hidden markov models for complex action recognition. *IEEE Conference on Computer Vision and Pattern Recognition* (pp. 994–999). San Juan, Puerto Rico: IEEE Press.

Cassandra, A., Kaelbling, L. P., & Littman, M. L. (1994). Acting optimally in partially observable stochastic domains. *Proceedings of the Twelfth National Conference on Artificial Intelligence.* Seattle, WA.

Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. *Proceedings of the Tenth International Conference on Artificial Intelligence* (pp. 183–188). San Jose, California: AAAI Press.

Dempster, A. P., Laird, N. M. & Rubin, D. B. (1977). Maximum-likelihood from incomplete data via the EM algorithm. *J. Royal Statist. Soc. Ser. B., 39.*

Dietterich, T. G. (2000). An overview of MAXQ hierarchical reinforcement learning. *Proceedings of the Symposium on Abstraction, Reformulation and Approximation SARA 2000*, Lecture Notes in Artificial Intelligence. Springer Verlag, New York.

Digney, B. (1996). Emergent hierarchical control structures: Learning reactive/hierarchical relationships in reinforcement environments. *From Animals to Animats 4: The Fourth Conference on Simulation of Adaptive Behavior.* MIT Press.

Fine, S. Singer, Y., & Tishby, N. (1998). The Hierarchical Hidden Markov Model: Analysis and Applications. *Machine Learning, 32*(1).

Gomez, G. F. & Miikulainen, R. (1999). Solving Non-Markovian Control Tasks with Neuroevolution. *In Proceedings of the 16th International Joint Conference on Artificial Intelligence* (pp. 1356-1361). Denver, CO: Morgan Kaufmann.

Hauskrecht, M. (2000). Value-function approximations for partially observable Markov decision processes. *Journal of Artificial Intelligence Research, 13,* 33–94.

Hernandez, N., & Mahadevan, S. (2001). Hierarchical memory-based reinforcement learning. *Proceedings of Neural Information Processing Systems 13,* (pp. 1047–1053).

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

Lanzi, P. L. (1997). *Solving Problems in Partially Observable Environments with Classier Systems (Experiments on Adding Memory to XCS).* Technical Report 97.45, Politecnico di Milano. Department of Electronic Engineering and Information Sciences

Lin, L., & Mitchell, T. (1992). *Memory approaches to reinforcement learning in non-Markovian domains.* Technical Report, Carnegie Mellon, Pittsburgh, PA.

Lin, L., & Mitchell, T. (1993). Reinforcement learning with hidden states. *From animals to animats 2: Proceedings of the second international conference on simulation of adaptive behavior* (pp. 271–280). Cambridge, MA: MIT Press.

Littman, M. L., Cassandra, A. R., & Kaelbling, L. P. (1995). Learning policies for partially observable environments: Scaling up. *Proceedings of the Twelfth International Conference on Machine Learning* (pp. 362–370). San Francisco: Morgan Kaufmann.

Loch, J., & Singh, S. (1998). Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes. *The Proceedings of the Fifteenth International Machine Learning Conference* (pp. 141–150). San Francisco: Morgan Kaufmann.

McCallum, R. A. (1993). Overcoming incomplete perception with utile distinction memory. *The Proceedings of the Tenth International Conference on Machine Learning* (pp. 190–196). San Francisco: Morgan Kaufmann.

McCallum, R. A. (1995a). Instance-Based Utile Distinctions for Reinforcement Learning with Hidden State. *The Proceedings of the Twelfth International Conference on Machine Learning* (pp. 387–395).

McCallum, R. A. (1995b). *Reinforcement Learning with Selective Attention and Hidden State*. Doctoral dissertation, Department of Computer Science, University of Rochester.

McGovern, A. (2002). *Autonomous Discovery of Temporal Abstractions from Interaction with an Environment.* Doctoral dissertation, University of Massachussetts.

Murphy, K., & Paskin, M. (2001). Linear time inference in hierarchical HMMs. *Proceedings of Neural Information Processing Systems 2001.*

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

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE, 77(2)* (pp. 257–286).

Sondik, E. J. (1971), *The optimal control of partially observable Markov decision processes.* Doctoral dissertation. Stanford University, Stanford, CA.

Sondik, E. J. (1978), The optimal control of partially observable Markov decision processes over the infinite horizon. Discounted costs. *Operations Research, 26,* 282–304.

Sun, R., & Sessions, C. (2000). Self-segmentation of sequences: automatic formation of hierarchies of sequential behaviors. *IEEE Transactions on Systems, Man, and Cybernetics: Part B Cybernetics, Vol.30, No.3*, (pp. 403–418).

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

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence, vol. 112*, pp. 181–211.

Theocharous, G. (2002). *Hierarchical Learning and Planning in Partially observable Markov Decision Processes.* Doctoral dissertation, Michigan State University.

Thrun, S. (2000). Monte carlo POMDPs. *Advances in Neural Information Processing Systems 12*, pp 1064-1070. MIT Press.

Tsitsiklis, J. N. (1994). Asynchronous stochastic approximation and Q-learning. *Machine Learning, 16*(3).

Whitehead, S. D., & Ballard, D. H. (1991). Learning to perceive and act by trial and error. *Machine Learning, 7(1)*, 45–83.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards.* Doctoral Dissertation, King's College, Cambradge, England.

Watkins, C. J. C. H. & Dayan, P. (1992). Q-learning. *Machine Learning.* 8:279-292.

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

Zhang, N. L., & Liu, W. (1996). *Planning in stochastic domains: problem characteristics and approximation.* Technical report, Department of Computer Science, Hong Kong University of Science and Technology.

Zhong, S., & Ghosh, J. (2001). *A New Formulation Coupled Hidden Markov Models*. Technical Report, University of Texas, Austin.