Reinforcement Learning with Selective Perception and Hidden State

by

Andrew Kachites McCallum

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Dana Ballard

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

1995, revised 1996

Curriculum Vitae

Andrew McCallum was born April 2, 1967, near Boston, Massachusetts. He grew up in Beaconsfield, England and Raleigh, North Carolina, eventually studying at The North Carolina School of Science and Mathematics, a nationally acclaimed public high school.

He attended Dartmouth College from 1985 to 1989, and graduated *summa cum laude*, with a B.A. in Computer Science. While at Dartmouth he performed senior projects with Professor Barry Fagan, programming a SIMD Connection Machine to find large prime numbers, and with Professor Peter Sandon, helping redesign the department's course in Artificial Intelligence.

After undergraduate school, the author spent one year doing research in machine learning at Biomedical Information Communication Center, Oregon Health Science University, in Portland, Oregon. There, it was his privilege to build an amicable and productive relationship with Dr. Kent Spackman. McCallum worked on inductive learning for medical diagnosis, using both artificial neural networks and genetic algorithms. This work culminated in the publication of his first paper, in the Proceedings of the Seventh International Machine Learning Conference. He also wrote reams of code for the Center's "Hospital of the Future" physicians workstation, extending NeXT Computer's object-oriented user interface toolkit to create custom human-computer interface objects for displaying medical data.

In 1990, he entered the Ph.D. program at the University of Rochester, and in 1992 received an M.Sc. in Computer Science. At Rochester, he published both in operating systems and machine learning. The author became interested in the issues of reinforcement learning with selective perception while working with Professor Dana Ballard and Steve Whitehead. Research in selective attention led the author directly to interest in problems with hidden state, which he continued to study under the guidance of Dana Ballard.

McCallum wrote the Reinforcement Learning Toolkit, a publicly available software library that makes it easy to test various reinforcement learning algorithms in different environments with different sensory-motor systems. Parts of the library turned into the GNU Objective C Class Library and GNUstep, for which McCallum is the project leader. He also contributed to several other Free Software Foundation projects, including GCC, Emacs and Guile.

He wrote two virtual reality simulators, the second on an SGI RealityEngine^2 machine, using SGI's *Performer* library. The simulator, called "Persia," has been used to build simulated environments of everything from gazelles stampeding on the African plain to traffic-congested four-lane highways. The code is publicly available and is the main software engine for the virtual reality lab's study of human eye movements during complex, natural tasks.

In July 1995, he married Donna Kachites. The two now live and bake bread together in Pittsburgh Pennsylvania's Squirrel Hill neighborhood.

Acknowledgments

Six years ago, I could not have predicted how grateful I would be to both Peter Sandon and Steve Whitehead for, respectively, pointing and beckoning me to the University of Rochester. The University of Rochester Computer Science department is a gem—small, yet sparkling with creativity and warmth. Its department-wide shared funding practice provides a foundation on which collegiality and good research thrive.

I thank everyone here—faculty, students and staff—for an extraordinary education.

My first thanks go to my advisor, Dana Ballard, for giving me the freedom and encouragement to pursue my research dreams, and for being patient and supportive even when those dreams were as vague and disconnected as night-time illusions. What I may have missed due to the famous triple-booked time slots, was more than made up for by our mind-meld research meetings at the Go board after an intense game. His ability to "see the big picture" and his enthusiasm are traits I'll always strive for.

Dana taught me enumerable lessons about research and life, even though many times I didn't realize the wisdom behind a seemingly whimsical remark until days, weeks or even years later. I have the feeling that, in years to come, I will continue to be enriched by latent seeds of wisdom he has planted in my head.

I also owe a debt of gratitude to others on my committee. I thank Chris Brown for his openness, his breadth of knowledge of the literature, his incredible ability to make detailed comments on papers with 48 hour turn-around time, his straight talk, wild French and hallway announcements. Chris, more than anyone, (except perhaps Jill, the department administrator), holds the department together. I thank Randal Nelson for tough questions, and Mary Hayhoe for keen insights from a new perspective, and for her supportive words. I thank Leslie Kaelbling for reaching out to a young researcher during a walk in the streets of Aberdeen, and for sharing many insightful conversations over conference lunch tables. Although we haven't gotten around to collaborative research paper yet, I do remember fondly collaborating on some delicious Chicken Florentine and some great recorder-and-piano duets. I also am extremely grateful for her in-depth reading and thoughtful comments on an earlier draft of this dissertation.

My fellow students at Rochester made the impossible possible. I thank Jonas Karlsson, who was like a brother to me from day one; Jeff Schneider, who heard countless late-night, hair-brained ideas and always had insightful comments; Bob Wisniewski for Go games when they were needed most; house mates Virginia de Sa, Leonidas Kontothanasis, Bob, Bill Garrett and Raj Rao for hilarious times during early morning sweeping, late night snow shoveling, and even later games of 'Ghost.' There is not space here to thank all the other students whose companionship enriched my life here, but I would especially like to mention Lambert Wixson, Mark Crovella, Jack Veenstra, Justinian Rosca, Eric Ringger, Colm O'Riain and Galen Hunt. I was privileged to share time and space over the years with several fabulous officemates. George Ferguson provided technical and social wisdom in ways only a wry Canadian can; Raj brought half the library to our office; Anastassia Ailamaki showed such excitement at seeing her first snowfall, I'll never look at it in quite the same way again.

Rochester has the best staff any department could ask for. I thank Pat for her smiles, Marty for her photography, Peggy for her endless patience, Ray for fixing my glasses, Jill for driving me to the emergency room, Jim for putting up with the extra backups, Brad for letting me borrow the keys, Luid for countless favors performed on short notice, Peg for being my partner in the "where's Dana" game, and Tim for being a friend.

It has been my pleasure to work with several undergraduate advisees over the years: Mia Stern, Chandler Chao, Jen Alexander, Matt Hohlfeld, and Josh Richardson. I especially thank Jen for being willing to wade into my 30,000 lines of reinforcement learning code, and Matt for hacking on the SGI late into the night.

Research buddies from other institutions—among them Michael Littman, Rich Caruana, Sebastian Thrun, Justin Boyan, Andrew Moore, Satinder Singh and Rich Sutton—shared with me both insight and wild times. In addition to the heated discussion and shared research excitement, I'll never forget skiing thrills at 10,000 feet, bushwhacking on snowshoes, and views of Jupiter's moons through binoculars from a wooded hillside.

This work would have been significantly more difficult without the Free Software Foundation, whose tools always came through for me without fail. I thank Jim Wilson at Cygnus Support for advice on GCC, and Richard Stallman for several helpful and patient conversations.

Gratitude with a wholly increased order of magnitude goes to my family.

My parents always gave me boundless and unconditional love, encouragement to follow whatever path I choose, and an appreciation for intellectual query. My gratitude is beyond words.

Rebecca, both my sibling and my dear friend, has seen me through all this. I thank her for her comradery, steadfastness, companionship and love.

And lastly, my thanks to Donna, my juggling partner, friend, dance partner, love, and wife. I thank her for her untiring love, encouragement, patience and wise words. I could not have made it without her. Thank you for the ocean.

This material is based upon work supported by NSF under Grant no. IRI-8903582 and by NIH/PHS under Grant no. 1 R24 RR06853-02.

Abstract

Reinforcement learning is a machine learning framework in which an agent manipulates its environment through a series of actions, and in response to each action, receives a reward value. The agent stores its knowledge about how to choose reward-maximizing actions in a mapping from agent-internal states to actions.

Agents often struggle with two opposite, yet intertwined, problems regarding their internal state space. First, the agent's state space may have "too many distinctions"—meaning that an abundance of perceptual data has resulted in a state space so large that it overwhelms the agent's limited resources for computation, storage and learning experience. This problem can often be solved if the agent uses *selective perception* to prune away unnecessary distinctions, and focus its attention only certain features. Second, even though there are "too many distinctions," the agent's state space may simultaneously contain "too few distinctions"—meaning that perceptual limitations, (such as field of view, acuity and occlusions), have temporarily hidden crucial features of the environment from the agent. This problem, called *hidden state*, can often be solved by using memory of features from previous views to augment the agent's perceptual inputs.

This dissertation presents algorithms that use selective perception and short-term memory to simultaneously prune and augment the state space provided by the agent's perceptual inputs. During learning, the agent selects task-relevant state distinctions with a *utile distinction test* that uses robust statistics to determine when a distinction helps the agent predict reward. The dissertation also advocates using instance-based (or "memory-based") learning for making efficient use of accumulated experience, and using a tree structure to hold variable-length memories. Four new algorithms are shown to perform a variety of tasks well—in some cases with more than an order-of-magnitude better performance than previous algorithms.

Keywords:

Machine Learning, Reinforcement Learning, Hidden State, Selective Perception, Selective Attention, Short-term Memory, State Identification, Utile Distinctions.

Memory-based Learning, Instance-based Learning, K-Nearest Neighbor (KNN), Model-free Reinforcement Learning.

State Aggregation, Factored Representation, Factored State Representation, Structured Policy Representation, Structured State Representation, Value Function Approximation, Variable-Resolution State, Tree-structured Policies, Model-based Reinforcement Learning.

Eye-movements, Fovea, Deictic Representations, Embodiment, Embodied Cognition.

Highway Driving, Block Stacking, Robot Navigation.

Forward

by Dana H. Ballard

Developing programs is simple when done by trained programmers, but how does the human brain learn its own programs? This question is at the center of the intersection of two fields, machine learning and computational neuroscience. Experimental evidence from animal studies now strongly suggests that the brain uses a secondary reward mechanism to denote good sequences of actions. This reward system is based on the neurotransmitter dopamine. The most striking recent work is by Wolfram Schultz from Zurich. Schultz has shown that a monkey's dopamanergic cells are activated at the beginning of a sequence of actions that lead to reward such as getting an apple. When the sequence of actions is changed, then so is the dopamanergic reward. It is activated at the beginning of the changed program.

Secondary reward has also been the focus of machine learning. One of the most important papers is by Barto and Sutton who described a detailed algorithm that could learn by using a secondary reward mechanism. This work essentially started the machine learning subfield of reinforcement learning and has lead to many faster variants of its original dynamic-programming based algorithm such as Q-learning and Temporal-difference (TD) learning. However one problem that these algorithms have failed to deal with is the representation of the problem itself. In reinforcement learning a problem is described in terms of a *state space* of all possible possible partial solutions to the problem.(For example in backgammon these would be board descriptions). *Actions* (such as the selection of pieces to move in backgammon) are ways of getting from one state to another. A successful solution consists of a sequence of actions that represent a path through the state space. Almost all previous reinforcement learning algorithms have depended on the programmer to pre-select the state space and actions, a very unrealistic possibility for large problems, and those which have attempted to define the state space algorithmically have exhibited very modest performance.

Andrew McCallum's dissertation defines a reinforcement learning algorithm that develops the appropriate part of the state space en route to solving its problem. It is overwhelmingly better than other reinforcement learning algorithms on the standard test problems that are used by the machine learning community and in addition it has solved a much larger problem of developing a controller for a simulated highway driving problem.

The major innovation of the dissertation is that it selects the state space of the problem from candidates by using a test based on secondary reward. The state space description is extended if that extension will lead to a distinction in rewards. This allows the state space description to be of variable size and to vary dynamically with experience. The net result is that problem sizes that can be tackled are now two to three orders of magnitude larger than that was previously possible.

The formalism for choosing the state space of the algorithm is that of *Hidden Markov Models (HMM's).* This is a very general formalism that has found particularly wide application in speech recognition. Another innovation is that the HMM states are kept in a tree. The tree data structure has the very important feature that it allows the decisions of the algorithm in different parts of the state space to be based on different *order* HMM's. Previously algorithms predominantly used fixed order HMM's, which are much more inefficient.

In summary this dissertation is a stand-out. It represents sparklingly innovative work that is important for our understanding of both machine learning and neuroscience. It has already made a significant impact in the reinforcement learning community and its importance is increasing as it becomes known to wider audiences.

Table of Contents

List of Figures

List of Tables

1 Introduction

Chapter Outline

This introductory chapter defines the terms "selective perception," "hidden state" and "reinforcement learning." It discusses the need for task-dependent representations, and then outlines the contributions of the dissertation.

Both this chapter and the next serve as introductions to the ideas and technical material of this dissertation. This chapter is intended for a general audience; the next chapter is aimed specifically at those readers who are already familiar with the issues of learning agents.

1.1 Selective Perception and Hidden State

How can an agent successfully interact with a complex environment when the agent's perceptual resources for sensing that environment are so limited?

Agents that interact with their environment through sensors and effectors often suffer from two opposite types of perceptual limitations. First, an agent can have "too much sensory data," meaning that the sensors provide so much raw data that the agent cannot possibly process all of it at once. In this case, an agent can use *selective perception* to focus its limited computational resources on processing only certain features.

Second, even though the sensors provide so much data, the agent may also have "too little sensory data," meaning that perceptual limitations, such as field of view, acuity and occlusions, can hide crucial features of the environment from the agent. This problem is called *hidden state*, and can often be solved by using context, or short-term memory to remember features that were available in previous views.

For example, consider the perceptual challenges of driving in city traffic. At each instant, the eyes of the driver produce an abundance of sensory data. The driver's retinal image may contain enough raw data to calculate the distance to the car in front, the color of the traffic light ahead, the position of a pedestrian waiting at the crosswalk, the type of clouds in the sky, the material composition of the car's visor, and the architectural style of the skyscraper to the right. However, the driver's limited computational resources would be overloaded if the driver attempted to attend to all these features at once. The driver should select some subset of these features—being sure to choose those features that are relevant to the task at hand.

While the driver's sensors provide too much data, they may also, simultaneously, provide too little data. Features of the world that are crucial to the driver's next choice of action may be hidden by perceptual limitations. Limited field of view may prevent the driver from seeing the car in the left side-street while the driver is looking at the traffic light. Limited acuity may prevent the driver from reading a street sign in the rear-view mirror. The bottom of the car may occlude a painted arrow on the asphalt below the car. When some of the relevant world features are missing from the current sensory snapshot, the driver can often decide what to do next using short-term memory of features from previous snapshots.

The two seemingly opposite problems of "too much sensory data" and "too little sensory data" are actually intimately related.

Selective perception can be seen as "creating hidden state on purpose"—the agent chooses to *hide* irrelevant features of perception from itself, thus selecting only what deserves attention.

Hidden state, when solved using short-term memory, can be seen as a problem of selective attention—the agent cannot possibly afford to remember everything; it must *select* what to remember and what to forget.

In both cases, the agent must choose which features—from present or past sensory data—the agent will attend to and which it will ignore.

What does it mean for the agent to attend to a particular feature? It means that the agent distinguishes between situations in which that feature is absent and situations in which that feature is present. This is why "attending to a feature" is also called "making a distinction." The cross-product of all distinctions chosen by the agent comprise the *agent-internal state space.*

With both selective perception and short-term memory for hidden state the agent must find those distinctions that are relevant to the agent's task at hand.

1.2 Reinforcement Learning

Reinforcement learning is a particularly clear framework in which to study the question "What is relevant to the task at hand?" because in reinforcement learning the agent's task is so clearly defined.

Reinforcement learning [Minsky, 1954; Barto *et al.*, 1983; Sutton, 1984; Watkins, 1989; Kaelbling, 1990a; Whitehead, 1992; Kaelbling *et al.*, 1995], as studied in the machine learning community, is a formal mathematical framework in which an agent manipulates its environment through a series of actions, and in response to each action, receives a reward value. An agent stores its knowledge about how to choose reward-maximizing actions in a mapping from agent-internal states to actions. In essence, the agent's "task" is to maximize its reward over time. Good task performance is precisely and mathematically defined by the reward values.

Those who are not familiar with reinforcement learning in the context of machine learning may be familiar with reinforcement learning from psychology. In the field of psychology, reinforcement learning has been often relegated to a fringe branch of learning that addresses only simple non-cognitive behaviors [Pavlov, 1923]. However, in the machine learning community, reinforcement learning has been used to solve many complex tasks normally thought of as quite cognitive. For example, a reinforcement learning algorithm has performed space shuttle pre-launch scheduling better than NASA experts [Zhang and Dietterich, 1995]; a reinforcement learning algorithm has learned to play backgammon, and now beats world-class human players [Tesauro, 1994]; a reinforcement learning algorithm has learned to perform elevator motion planning better than the schemes devised by Otis engineers [Crites and Barto, 1995]. There is reason to believe that reinforcement learning is a broadly applicable framework for problem solving in mammals. For example, neuro-biologists have found what they believe may be the neural-correlate of reinforcement signals in the brain [Montague *et al.*, 1995]; and reinforcement-like emotional cues are hypothesized to be related to high-level decision-making [Damazio, 1994].

The little-discussed black magic of many reinforcement learning applications is the delicate engineering of the agent's state distinctions. The agent designer builds and tweaks the agent's internal state space so that it represents those distinctions that are relevant to the current task. But this is difficult, and the designer can often fail. How can the designer know what is important? In order to know which distinctions are necessary, the designer needs to know how the agent will solve the task, but that is exactly what we want the agent to learn! The designer is all too likely to get it wrong—either by giving the agent too many distinctions, or too few.

1.3 Learning Task-Relevant Distinctions

Instead of having the agent designer fix the agent-internal state representation before learning, we should have the agent learn a task-dependent state representation. The agent can do so by selecting a set of relevant features during training. Thus, although the agent designer defines a fixed sensory system for the agent, the learning algorithm can correct for "errors" in either direction—by using selective perception to prune out distinctions that are irrelevant to the task, and by using short-term memory to augment the state space when there is task-impeding hidden state.

Why is it important to choose only the relevant distinctions? If the agent makes too few distinctions, then the agent will suffer from hidden state and fail to perform well—in particular, it will collapse states that require different behavior. On the other hand, if the agent makes too many distinctions, then the agent will not only waste computer storage and computation, but, more importantly, will also generalize poorly and require more experience to learn separately the outcomes of actions from the needlessly distinguished states.

Learning which features are relevant to a task is a large part of learning any skill. Novice drivers find city driving extremely difficult, in large part because they do not yet know where to attend. They may flit their attention all over the scene, yet fail to successfully attend to relevant features because either, amidst all those gaze changes they still miss a relevant feature, or because they spread their attentional resources so thinly that they lose track of the important features they may have seen.

1.4 This Work in Perspective

This dissertation studies the problem of learning to allot limited perceptual resources and select task-relevant features during the performance of multi-step reinforcement learning tasks.

The presented algorithms select from among features of the current sensor values, and, when context over time is relevant, also select from among features of past sensor values.

Much research work has struggled to solve problems encountered when reinforcement learning is applied to agents that use a *fixed* set of features. Many studies have explored techniques for speeding up or otherwise improving performance on tasks that have more features than can be handled efficiently in simple traditional ways, (*e.g.* [Sutton, 1990a; Lin, 1991a; Singh, 1992]). Several studies have explored techniques for performing as well as possible when using too few features (*e.g.* [Whitehead, 1992; Littman, 1994a; Jaakkola *et al.*, 1995]). There has been relatively little work, however, on techniques for avoiding these problems by automatically selecting a *variable* number of features—by both pruning away features that aren't relevant and by augmenting the number of features by looking into the past.[1] This dissertation contributes to this little-studied area.

1.5 Contributions

I divide the contributions of this dissertation into two categories: first, what I call *soft contributions*—the viewpoints, taxonomies, goals and biases that form the launching point for new ideas; and second, what I call *hard contributions*—the algorithms, proofs and experimental results that are the technical fruit of these soft contributions.

[1]Notable exceptions are discussed in Chapter 3. Previous work in which the agent prunes away irrelevant features is discussed in section 3.3. Previous work in which the agent augments its state space by looking into the past is discussed in section 3.2. I am not aware of previous work in which the agent both prunes and augments.

The importance of soft contributions is often underestimated. Selecting beneficial biases, choosing a viewpoint on the problem and defining the problem all lay the important foundation upon which technical contributions are pieced together. A significant aspect of the soft contribution made by a dissertation is the "story it weaves." The threads of the story are the themes and ideas that, while not necessarily unique to the author, are woven together to make a unique juxtaposition, bringing together a new pattern that was not previously visible.

Some of the soft contributions of this dissertation are described as follows.

- There are different types of hidden state. I provide a two-dimensional taxonomy of hidden state that recognizes both the "degree of correlations" and "degree of hiddenness" of world state features. I also introduce the terminology "Markov with respect to," and explain the usefulness of this idea for categorizing and handling hidden state. (Chapter 2, page 14.)

- Selective perception and hidden state are intimately related. Selective perception can be seen as "purposefully creating hidden state." I explain that these seemingly opposite problems—in which, on the one hand we have "too much sensory data," and on the other hand we have "too little sensory data"—are actually quite interrelated. I present solutions to these problems that are also interrelated. (Chapter 2, page 22.)

- I define the term "utile distinctions" as the state distinctions that help predict reward. I discuss the importance of utile distinctions to building task-dependent internal state representations. When the task is simple, we would like learning to be simple, even when the environment is complex. Utile distinctions help the agent achieve this. (Chapter 2, page 17.)

- I identify the key difficulties of reinforcement learning with hidden state. (Chapter 2, page 22.)

- I explain the commonalities between learning with hidden state and learning in a continuous state space. Instance-based (memory-based) learning is a paradigm in which the learner records raw experience. I discuss the ways in which instance-based learning is particularly well-suited to both continuous spaces and hidden state. (Chapter 5, page 58.)

- For uncovering hidden state, I choose to use models based on finite-length memories, as opposed to the relatively more expressive models based on finite state machines with loops. The former are much easier to learn, and I show that they are adequate for solving many complex tasks. I also, however, discuss the situations in which these simpler models are likely to perform poorly. (Chapter 6, page 72.)

The technical contributions of this dissertation are summarized as follows.

- I present a simple proof (by counter example) that the state distinctions required for representing the optimal policy are not necessarily sufficient for calculating the optimal policy. The example grew out of discussions with Kaelbling [Kaelbling, 1995]. (Chapter 4.)

- I explain a simple statistical technique for finding utile distinctions, and I present a novel algorithm that makes utile distinctions for the purpose of uncovering hidden state. The algorithm, called Utile Distinction Memory (UDM), is a modification of the Baum-Welch hidden-Markov-model-learning algorithm [Rabiner, 1989] that learns to split and merge states dynamically, contingent on the utile distinction statistical test. (Chapter 4.)

- I present a novel modification of the k-nearest neighbor algorithm that finds neighbors in "sequence space" instead of "Euclidean space." The resulting reinforcement learning algorithm, called Nearest Sequence Memory (NSM), learns to solve tasks with hidden state an order of magnitude faster than several other algorithms. (Chapter 5.)

- I present an adaptation of Prediction Suffix Tree Learning [Ron *et al.*, 1994] to reinforcement learning. The two key modifications entail having the algorithm generate its own training data and adding actions and reward to the model. The new algorithm, called Utile Suffix Memory (USM), can also be seen as a combination of the ideas from UDM and NSM. I present a comparison of this new algorithm with Whitehead's Lion algorithm [Whitehead, 1992] for solving tasks using visual routines, and explain how Utile Suffix Memory affords a new approach that is both more powerful and less wasteful of perceptual resources. (Chapter 6).

- I introduce, U-Tree, a new algorithm, based on Utile Suffix Memory, that makes both utile *memory* distinctions and *perceptual* distinctions. Thus, the algorithm can uncover hidden state and can perform selective perception. It does this using what is termed a *Factored State Representation* (or *Structured State Representation*), in space of state variables and in the space of histories.

  I demonstrate the algorithm solving a complex driving task involving over 21,000 world states, 2,500 percepts, task-impeding hidden state, stochasticity, and time-pressure. I believe that this task is more difficult than any other task previously attempted by a hidden state reinforcement learning algorithm. (Chapter 7.)

- I discuss specific details of possible future enhancements, including: finding utile distinctions in continuous state spaces, finding utile distinctions in action space, using complex and biased distinctions, using smarter methods of exploration, and developing information-theoretic techniques for learning better distinction trees. (Chapter 8.)

1.6 Outline

Chapter 1 is intended to be an introduction for general audience. Chapter 2 serves as a more detailed introduction for people already familiar with learning agents. In it I explain the motivations and principles behind this work and provide important context for many of the design decisions that went into the algorithms presented in this dissertation.

Chapter 3 introduces basic technical foundations for reinforcement learning, Markov models, hidden state and selective perception. It also surveys previous work in reinforcement learning, hidden state and selective perception.

Chapters 4 through 7 comprise the body of the dissertation. In them I describe a succession of four algorithms—each of which builds on its predecessors. While each of the algorithms has unique positive features, the sequence can mostly be understood as a cumulative improvement. I choose to describe each of the algorithms, (instead of skipping to the last), not only because I want to explain each of their unique positive features, but also to illustrate the path of this research.

Utile Distinction Memory: Chapter 4 discusses the ideas behind a utile distinction test that can robustly separate noise from structure in order to find new state distinctions that help the agent increase its reward. This chapter presents the Utile Distinction Memory algorithm, a technique that learns utile distinctions using a partially observable Markov decision process—but it is slow to learn, and has trouble finding multi-step memories.

Nearest Sequence Memory: Chapter 5 introduces instance-based (or "memory-based") learning and discusses its special applicability to problems that involve learning new distinctions. The Nearest Sequence Memory algorithm is presented here. Unlike Utile Distinction Memory, Nearest Sequence Memory learns in a small number of steps, and it can easily discover long, multi-step memories. However, the algorithm does not make utile distinctions, and it does not deal with noise very well.

Utile Suffix Memory: Chapter 6 presents a type of finite state machine called a "suffix tree," shows how the model captures variable resolution memories, and explains how it is learned. The chapter presents the Utile Suffix Memory algorithm. This algorithm combines the best features of Utile Distinction Memory and Nearest Sequence Memory, resulting in an algorithm that is fast, builds multi-step memories, makes utile distinctions and is robust to noise. However, like its predecessors, it does not handle large perceptual spaces well because it only learns variable-resolution history distinctions, not variable-resolution perceptual distinctions.

U-Tree: Chapter 7 presents the first algorithm in this series that learns to select both history and perceptual distinctions, and is the culmination of this dissertation work. The algorithm, called U-Tree, is built on Utile Suffix Memory, but chooses from among both perceptual and history distinctions. This means that the algorithm can correct for both “too much sensory data” and “too little sensory data.”

Each of these four “algorithm” chapters also presents empirical results with the algorithm, and several times uses identical test environments in order to make direct comparisons to other algorithms.

The dissertation concludes with chapter 8, in which I discuss how this work can be applied, its strengths and limitations, and ideas for further improvements.

2 Motivations

Chapter Outline

While chapter 3 provides the technical background for the dissertation, this chapter provides the philosophical background. This discussion of motivations not only presents my views and biases at the outset, but also provides a structure in which to organize discussion of some related work.

I am motivated by the desire to build agents that learn to solve interesting tasks by interacting with their environment through sensors and effectors. I am less concerned with agents that do not learn, and with agents that are given complete models of their environment.

This dissertation is built on the following chain of principles:

- Learning closed-loop behaviors is useful;
- Selective perception provides the best interface to the world for closed-loop interaction;
- Many interfaces to the world suffer from hidden state—and selective perception often makes the hidden state problem worse;
- The non-Markov hidden state problem can be solved with memory;
- Selective perception and memory for hidden state are intimately related—and in both cases we want to make task-relevant distinctions;
- Learning to select perceptual features and use memory is difficult and requires special algorithms.

In addition, I believe that:

- Experience is often more expensive than computation and computer memory;

- Agents must be able to handle noisy perception, action and reward;
- Strict optimality is not always necessary.

In this chapter I explain and support each of these statements.

"Learning closed-loop behaviors is useful."

A closed-loop system is one in which the latest outputs of the system are used to help determine the next controlling inputs to the system [Luenberger, 1979]. Many tasks that we would like robots to perform are best managed by closed-loop control. For example: collision-free motion [Brooks, 1986], visual tracking [Coombs, 1992; Brown *et al.*, 1989], keeping a car on the road [Pomerleau, 1992], dextrous manipulation [Raibert and Craig, 1981; Pook and Ballard, 1992], steering a bicycle [Schneider, 1995] and navigation in traffic [Reece and Shafer, 1992]. These are all tasks that involve sequences of actions, where the effects of the actions are cumulative. Successful performance requires a closed-loop sense-act cycle because not all information relevant to the task can be perceived at the outset, and the effects of actions are uncertain.

Agents that learn behaviors build a mapping from the agent's internal state space to correct actions. We would prefer that our robots learn their behaviors rather than have them hard-coded because programming all the details by hand is tedious, we may not know the environment ahead of time, and we want the robot to adapt its behavior as the environment changes.

Supervised learning is often awkward because the teacher must provide all the correct responses in the agent's own unintuitive output space, such as vectors of joint torques. Learning techniques in which a teacher provides more intuitive feedback, such as reinforcement [Minsky, 1954; Watkins, 1989], or a distal error correction [Jordan and Rumelhart, 1990] are more desirable.

"Selective perception provides an efficient interface to the world."

For tasks involving closed-loop sequences of actions, *selective perception* is an efficient method for gathering relevant information from the world. An agent has selective perception if it can explicitly control which world features are used to determine its internal state.

This explicit control can be in the form of *overt attention*, which involves physically moving the sensors, or it be in the form of *covert attention*, which involves changing what computations are used to extract information from the current or past raw sensor data. Eye movements and camera movements are examples of overt attentional shifts; shifting concentration from judging the shape of an item to judging the color of the item is an example of a covert attentional shift.

The act of shifting attention is called a *perceptual action*. This is in contrast to a *manipulative action*, which changes some part of the world external to the agent. Although the terms "manipulative action," "perceptual action," "overt attention" and

Throwing a ball.
Lifting a box.
Moving down a hallway.
Touching an object.
Tasting an object.
Moving an eye.
Shifting concentration from judging shape to judging color.

Table 2.1: Examples in the continuum from manipulative actions to covert selective attention.

"covert attention" are useful, we cannot draw firm boundaries between them—they form a continuum from actions that are primarily manipulative of world state, such as lifting a box, to actions that primarily change perceptual processing, such as shifting concentration from the words written on a piece of paper to the color of the paper. Note that, even at the extremes of either side of the continuum, we can see elements of the other side. Returning to our examples: lifting the box is also a perceptual action in that it uncovers what was underneath the box; shifting concentration does change the state of the world in as much as the agent's brain is a physical element of the world. (Examples from this continuum are depicted in table 2.1.)

In keeping with this continuum, the algorithms presented in this dissertation require no explicit distinction between perceptual and manipulative actions.

Selective perception has been most intensely studied in the context of computer vision. *Selective vision* [Burt, 1988; Brown, 1992] is intimately related to *active perception* [Aloimonos *et al.*, 1988], *animate perception* [Ballard, 1989; Ballard and Brown, 1992], and *purposive perception* [Garvey, 1976]. Active and animate perception emphasize the computational advantages of camera movement, such as looking around occlusions, moving closer to an object to obtain better resolution, and translating to get depth-from-motion cues. Purposive vision emphasizes efficient task-dependent sensing. Selective vision emphasizes the redirection of attention. Most people would comfortably refer to eye movements using any of the terms.

Advantages of Selective Perception

Let us use an example to illustrate some advantages of selective perception. A robot trying to put a cup on a saucer need not perceive and represent the entire table setting. If it has an active vision system it may recognize only the object in its center of view. The robot would move its cameras to find and focus on the cup, pick it up, move its eyes to find the saucer, then place the cup on the saucer.

There are three important advantages of selective perception to notice in this example. First, the agent has used its sensing resources efficiently; it did not do the calculations necessary to recognize all the irrelevant plates, napkins, forks and knives. Second, by selecting just a few features by which to make state distinctions, the agent has dramatically reduced the size of its internal state space, and thus reduced its storage and computational load. Third, and most important to learning, the smaller state space created by selective perception has provided the robot with a powerful generalization technique.

Generalization is a learner's ability to use past experience to perform successfully in new, different situations; without it, learning is nothing more than rote table-lookup. Typically, generalization clusters training examples into equivalence classes by ignoring some aspect of the input, (or it interpolates between training examples). Successful generalization usually involves knowing which aspects of the current situation are important to finding the closest applicable past experience, and which aspects should be ignored. Because the agent's selective perception system ignores parts of the world state, different world states may map to the same agent perception, and thus become members of the same equivalence class. By directing its sensors towards only those parts of the world that are relevant to the current task, the agent has direct, precise control over how it generalizes.

Generalization is also related to the notion of an invariant, such as the geometrical invariants uses in computer vision [Forsyth *et al.*, 1991]. When the agent finds itself in different world states in which it is supposed to execute the same action and maps the world states into the same percept, the agent has, in a way, found a useful invariant. An invariant is a property that is not changed by specific physical transformations. For instance, the area of a two-dimensional shape is invariant to rotation in the visual plane. If the set of two-dimensional objects the machine is trying to recognize all have different areas, using this invariant property would make the object recognition task easy—we have reduced the search space from the many-dimensional space of pixel intensities to the one-dimensional space of shape areas. Returning to the invariants that selective perception provides in the table-setting domain, we would say that the robot's selective perception policy is invariant to the presence of napkins, forks, knives and plates. Because of this many-to-one mapping (many different table settings to one focused perception) the robot will be able to perform the cup-on-saucer task without an increasingly complicated policy on cluttered as well as uncluttered tables.

This many-to-one mapping from world states to perceptions can also go the other direction, (many perceptions for one world state)—that is, with selective perception, different agent perceptions can result from the same world state. For instance, given the same table setting, the robot's perception will obviously differ if the robot shifts its gaze. Thus, an agent with selective perception actually works with a many-to-many mapping between agent perceptions and states of the world. Whitehead calls this many-to-many mapping between states of the world and percepts of the agent *perceptual aliasing* [Whitehead and Ballard, 1991a].

Perceptual aliasing is both a blessing and a curse. It is a blessing because it can provide useful invariants by representing as equivalent world states in which the same action is required. It is a curse because it can also confound different world states in which different actions are required. Perceptual aliasing provides powerful generalization, but it can also over-generalize. The trick is to use the agent's resources intelligently to make the required distinctions between perceptually aliased states while keeping the useful generalization provided by the other aliasing. Selective perception gives the agent more freedom over how it perceives the world, but with increased freedom must come more responsibility [Clinton, 1993].

"Many interfaces to the world suffer from hidden state, and selective perception often makes the hidden state problem worse."

The sensory systems of embedded agents are inherently limited. When an agent's sensory limitations hide features of the world from the agent's current perceptual snapshot, we say that the agent suffers from *hidden state*.

There are many reasons why important world features can be hidden from a robot's perception:

- sensors have limited range, acuity and field of view;
- two-dimensional retinas provide limited data about the three-dimensional world
- sensor noise can produce unreliable data;[1]
- occlusions hide areas from sensing;
- the robot may not be able to sense something without modifying or destroying it;
- limited funds and space prevent equipping the robot with all desired sensors;
- an exhaustible power supply deters the robot from using all sensors all the time;
- the robot has limited computational resources for turning raw sensor data into usable percepts.

Consider the following examples:

Repair A space station repair robot sees that it needs a replacement part and thrusts across the station towards a supply bin to get the new part. However, once at the bin, occlusions and limited range prevent the robot from still seeing the distant faulty part. Given only the percepts available from its position at the supply bin, the robot cannot know which part it needs.

Navigation A hospital delivery robot is making its way from the patient's room to the blood lab. Its inexpensive and limited perceptual apparatus consists of range-finding sensors pointing forward, back, right and left. Each sensor returns an estimate of the distance to a barrier in the sensor's direction. During its travel, the robot's limited sensors are not able to distinguish between many of the different

[1]Sensor noise is not usually thought of as non-Markovian hidden state, however, it can be, in as much as remembering the values from multiple readings and taking an average of them will result in better knowledge of world state.

hallway intersections. In some of the aliased intersections it should turn right and in others it should turn left. Given this confusion, the robot cannot reliably deliver the sample.

Driving A robot driver uses a vision system to navigate in traffic. In order to overcome limited computational power for extracting usable perceptual information from its raw image data, (*i.e.* the last perceptual limitation in the list above), the driver makes use of an active vision system [Ballard and Brown, 1992; Reece, 1992]; it has movable binocular cameras, each containing a fovea. The high resolution in the fovea is necessary for recognizing objects, but if the cameras had such fine-grained resolution over a broad field of view, the vision system would be inundated with far more pixels than its limited computational power could handle.

The driver should pass a car on the highway only when it is close to the preceding car and the blind spot is clear of cars. Unfortunately, addressing the computational limitation has intensified another limitation: limited field of view. The driver cannot see both the car in front and the blind spot at the same time. Since neither a close-by car ahead nor a clear blind spot alone are enough to tell the robot that it should pull into the passing lane, and since no matter how the robot redirects its active vision system, no immediate perception can include both inputs to the conjunction, the driver cannot decide to pass using immediate perception alone.

Perceptual limitations apply to agents both with and without selective perception. In the robot driving example above, the agent could redirect its attention by moving its sensors, and thus have selective perception; in the robot navigation example, the robot's sensors and perceptual computations were fixed. Both agents suffer from hidden state.

Types of hidden state

Think of the state of the world as being captured by a huge set of state variables. The agent, through its perceptual system, can have some knowledge about the values of some of these variables. I define *hidden state* as any world state information that is not provided by the current, immediate perceptual snapshot.

The term hidden state has been used quite inconsistently in the literature.[2] For example, some have used hidden state to mean only state information that is relevant to the current task, but not provided by the current percept. While this may be the most detrimental type of hidden state, there are other types of hidden state that often deserve thought and discussion. Thus, I prefer to give hidden state its broadest possible meaning, and then provide modifiers that discriminate between the different types of hidden state. (On page 16, I give further justification for my broad definition.)

I have defined hidden state relative to "world state," but no agent can possibly know the state of the entire world. Thus, in a way, all agents suffer from hidden state. This subsection introduces more specific terms, presents a useful taxonomy of hidden state,

[2]The author is included among the past inconsistent users.

discusses the concept of "task-relevant state" and more precisely defines the term *utile distinctions.*

The taxonomy of hidden state has two dimensions, one corresponding to each word in the term "hidden state." There is the degree of "hidden-ness," and there are the "state attributes" in question. The first is related to correlations with the past; the second is related to correlations with the future. A description of the items in each of these dimensions follows; the taxonomy appears in table 2.2.

	visible	non-Markov	invisible
state			
perception			
perceptual subset			
reward		$\star$	
inconsequential			

Table 2.2: A taxonomy of hidden state. The degree of "hidden-ness" is on the horizontal axis; the "state attributes" in question are on the vertical axis. The star marks "recoverable utile hidden state."

Consider the following scenario:

> Shaquille O'Neil has the basketball. He looks right, sees his teammate, who signals that he is ready for a pass; Shaquille looks left at the basket, fakes a shot at the basket, and then passes the ball to his teammate on the right without even looking.

Considering the moment Shaquille passes the ball, let us use this example to talk about some different kinds of hidden state.

Visible First of all, there is some state that is not hidden. In our scenario an example of this is the feature "Shaquille's distance to the basket." The distance is provided by the current outputs of the agent's perceptual system.

State that is immediately observable, *i.e.*, provided by the agent's current percept, I call *visible state.* Whether or not a state variable is part of visible state is a function of the world and the agent's perceptual system.

Non-Markov Features that are not immediately observable, but are correlated with past state, observations or actions are called *non-Markov hidden state*; I also call this *recoverable hidden state.*

A model[3] has the *Markov property.* if and only if knowledge of past model states does not help predict future model state.

[3]When I say "model" here, I am thinking of the agent-internal state space and its transition function

Much useful hidden state is non-Markov hidden state. For example, the fact that Shaquille's teammate is to the right and ready for a pass is non-Markov hidden state because that state is correlated with Shaquille having seen his teammate in a previous view.

Whether or not a state variable is part of non-Markov hidden state is a function of the world and the agent's perceptual system.

The agent can use memory of past features to distinguish between situations with differing non-Markov hidden state.

Invisible Then, finally, there may be world state that isn't correlated with anything the agent perceives or ever has perceived. I call this kind of hidden state *invisible hidden state.*

An example of invisible hidden state in our Shaquille scenario might be the position of a player behind Shaquille who made a sudden, unpredictable movement, or, more dramatically, what a shoemaker in Tibet is having for dinner.

Whether or not a state variable is part of invisible hidden state is a function of the world, and the agent's perceptual system. The agent cannot make state distinctions that discriminate between invisible hidden states.

For instance, if an agent's task involves color, but the agent can't perceive color, or anything correlated with color, then the agent is just plain out of luck—it has no possibility of uncovering that hidden state. In the agent's world, color doesn't exist. In some more common definitions of state, we would not be able to talk about color because it does not exist in the agent's world. Note, however, that *our* conception of world is not the same as the agent's; it is useful for us to talk about this state, point to it, understand why the agent is failing, and have a name for this kind of hidden state.

The previous dimension in the taxonomy of hidden state indicated the degree of hidden-ness. The second dimension in the taxonomy of hidden state indicates which state attributes are hidden.

Reward There may exists some state that is not immediately observable, but that is correlated with future reward. These state distinctions, when used to define a state model, provide a model that I term *Markov with respect to* reward.

By "Markov with respect to X," I mean that knowledge of past model states does not help predict future values of X. As an advantage over the traditional "Markov property" definition, this terminology, introduces the flexibility of choosing *what* it is that we care about predicting.

Traditionally, a model is said to have the *Markov property* when it is "Markov with respect to model state," i.e. a model has the Markov property if future state is conditionally independent of past states (and actions), given the current state (and action). We write this as follows:

$$\Pr(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ...s_0) = \Pr(s_{t+1}|s_t, a_t), \tag{2.1}$$

where s_t is the model state at time t, and a_t is the action chosen at time t, and subscripts $t-1$ and $t+1$ indicate steps in the past and future.

However, we can also talk about the model being "Markov with respect to reward," or "Markov with respect to perception" or "Markov with respect to a certain feature of perception."

A model is "Markov with respect to reward" if and only if future *reward* is conditionally independent of past states and actions, given the current state and action. We write:

$$\Pr(\mathbf{r}_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ...s_0) = \Pr(\mathbf{r}_{t+1}|s_t, a_t) \tag{2.2}$$

where $\mathbf{r}_{t+1}$ is the expected future discounted reward at time $t+1$.

Hidden state that is correlated with future reward would help the agent predict future rewards, and thus is useful for choosing actions. An example of this kind of hidden state in our scenario would be Shaquille's knowledge that his teammate is to the right and ready to receive a pass.

I call hidden state correlated with future reward "task dependent" or *utile hidden state*, where "utile" refers to "utility", which is another term for expected future reward.

Whether or not a state variable is part of utile hidden state is a function of the world, the agent's perceptual system and the agent's task (*i.e.* the agent's reward function) and the agent's policy. State distinctions that discriminate between utile hidden states are called *utile distinctions.*

For example, if

$$\Pr(\mathbf{r}_{t+1}|o_t, a_t, o_{t-1}) \neq \Pr(\mathbf{r}_{t+1}|o_t, a_t) \tag{2.3}$$

(where o_t is the agent's observation at time t and o_{t-1} is the agent's observation at time $t-1$), then o_{t-1} is *utile hidden state*, and incorporating o_{t-1} into the agent's state representation, s, would be making a *utile distinction.*

Note that there may be different sets of distinctions that result in a model that is Markov with respect to reward, and the sets may have different size. Note also that the property depends on the agent's policy; there is a discussion of this dependency in section 7.6.

Why is making only utile distinctions important? For the same reasons selective perception is important—when the agent makes more state distinctions than are relevant to the task at hand, it does not generalize well and makes inefficient use of storage, computation and experience.

Perception There can also be state that is correlated with future observations. These state distinctions would be needed to build a complete model of all the agent's perceivable world, but they do not help the agent perform its task.

In our scenario, this might be the shirt color of a person sitting in the stands. It is perceivable by Shaquille, but it completely unrelated to which action he should choose to help his team score points.

I call this state *perceptual hidden state.* Whether or not a state variable is part of perceptual hidden state is a function of the world and the agent's perceptual system. State distinctions that discriminate between perceptual hidden states are called *perceptual distinctions.*

Subset of Perception There is also state that is correlated with only a subset of future observations. This state would not be sufficient for building a complete model of all the agent's perceivable world, and it may include state that does not help the agent perform its task, but this state would provide a model that could predict certain features of perception.

If the agent designer knew that a certain subset of perceptual features were always correlated with future reward, then having the agent build a state model that is Markov with respect to this subset might help the agent find the task-improving state distinctions.

Agent State The traditional definition of the Markov property is not directly related to reward or perception—it is related to agent state. A model has the Markov property if knowledge of past model states does not help predict future model states. Thus, another category of hidden state is state that is correlated with future model states—whatever those model states really mean.

Inconsequential There is some world state that is "over the agent's head"—it is totally uncorrelated with anything the agent will ever see.

This category is analogous to invisible hidden state, but looking into the future instead of the past.

The most interesting type of hidden state is the task-dependent state that the agent can recover, namely, utile, non-Markov hidden state.

Selective perception often brings on hidden state

Although hidden state can appear without selective perception, I assert that hidden state is especially likely with selective perception because the defining aspect of selective perception is directable, focused (i.e. purposefully narrowed or specialized, and thus limited) attention.

Selective perception has an interesting relationship with hidden state. On the one hand, selective perception was created in response to perceptual limitations (especially limited computational resources and bounded space, energy and funds for all desired sensors.) It recognizes the limitations by dictating that the agent should not try to perceive everything at once; the agent must ration its resources by focusing its attention on just a few things at a time (See [Agre, 1985], for example). On the other hand, this focusing, this narrowing of perceptual attention by choice, makes the potential for hidden state worse by decreasing the effective field of view or by using only a subset of the available object recognizers.

For instance, a robot with a vision system built by the principles of selective perception may have a directable camera on a rotating head. The robot would have a limited field of view (it would not be able to see in front and behind it at the same time), but it could perceive other areas in sequence by executing the perceptual actions to move its eyes and head. Now consider a robot built without selective perception. In order to perceive the same region around itself, the robot may may have four cameras, one each pointing front, back, right and left. The robot would do the computation to interpret the images on all four cameras all the time. The resulting robot would run more slowly and/or require more resources than the selective-perception robot, but it would have less hidden state.

This last example demonstrates the way in which selective perception overcomes its purposefully limited focus: it redirects its attention over time. This technique, however, does not solve the hidden state problem because it is still the case that the perception *at any one instant* has limitations that hides aspects of the environment. Active, selective perception effectively takes tasks that are serial and makes them even *more* serial—by serializing perception.

Responding to the hidden state problem

The relevance of hidden state in the context of recent robotics research becomes clear when we consider the surge of interest in *reactive systems* [Brooks, 1986; Kaelbling, 1986; Brooks, 1987]. All types of agents must choose their next action using a mapping from the agent's internal state space to its choice of action. In what I call a "purely reactive" or "memory-less" agent, the agent's internal state space is *defined* as its current perception. When a purely reactive agent suffers from utile hidden state, its task performance will suffer. In a sense, reactive systems are closed loop systems taken to the extreme—they use *nothing* but the latest inputs from the environment to choose the next controlling inputs. They have the advantages of closed loop systems (they work well in environments where the results of actions are uncertain), but when the latest outputs aren't enough, they fail.

Anticipating this failure, many "reactive" robots are not actually purely reactive, but include some part of internal state space determined by inputs other than the current perception. [Brooks, 1991; Wixson and Ballard, 1991]

However, recent research in *learning* for robotics has strongly emphasized purely reactive agents [Maes and Brooks, 1990; Sutton, 1991; Mahadevan and Connell, 1991; Chapman and Kaelbling, 1991] Even past research on learning with selective perception has used a purely reactive approach. For example, in Whitehead's dissertation [Whitehead, 1992], the agent uses a selective perception system based on two deictic markers [Agre and Chapman, 1987] to perform manipulations in a block stacking task. Whitehead's Lion algorithm avoids some of the problems of perceptual aliasing, but it does not uncover hidden state (i.e. it would not solve tasks that require passing through a perceptually aliased state). Since the agent uses only current perception to choose its next action, the agent's perception space had to be carefully crafted to prevent task-impeding hidden state. The current perception included a bit that tested for vertically aligned block markers. Without understanding the agent's internal state space and the sequence of actions required to complete the task it may have been difficult to guess that the agent would need this perceptual bit to avoid hidden state.

Getting rid of hidden state means adding more immediate perception. This may or may not be possible given the limitations on perception listed on page 13. But more importantly, by adding more sensors, or by performing more computation on the raw sensor data the agent already has, we would be moving in the opposite direction from the compact, efficient robots that recognize inherent space, monetary, energy and computation limitations; and we would be moving further from the desirable "selective generalization" that gives the agent efficient control over what it perceives. What we want is some mechanism for efficiently providing the extra perceptual information only when it is required.

Here is my point: Even in cases where it is possible, it is both tricky and undesirable to create perceptual systems that avoid hidden state. We should develop methods for learning to perform tasks *in the presence of hidden state.*

"The non-Markov hidden state problem can be solved with memory."

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

The memory used to disambiguate hidden state must not be confused with mechanism used to store the agent's mapping from its internal state to actions (which might also reasonably be called memory). The first comprises part of the agent's internal state, the second uses that internal state to choose an action. For instance, in a neural network, the activations on the input layer define the internal state, while the weights provide the mapping from internal state to outputs. Those input layer activations may come from outside the network (corresponding to current perception), and they may also come in part from the activations of other units during the previous time step (corresponding to memory). To distinguish the internal state memory from the mapping memory, I sometimes refer to the first as "short-term memory." The mapping memory could conceivably be referred to as "long term memory."[4]

Returning to our examples above, consider how the agent might use short-term memory to uncover hidden state.

[4]This distinction may not actually be quite as simple as the above paragraph makes it sound. Consider an agent whose only memory is in its reactive action map—what I called "long-term" memory above. But let it have a very adaptive learning algorithm. Then it will have different policies in different parts of the world, thus using what I called "short-term" memory of the world to choose actions.

Repair The agent suffered from utile hidden state because, given only the percepts available from its position at the supply bin, the robot could not know which part it needed.

With short-term memory, however, the robot could remember which part was faulty, and pick the correct part.

Navigation The agent suffered from utile hidden state because it could not distinguish between two intersections, one in which it should turn left, another in which it should turn right.

With short-term memory, however, the robot could distinguish identical-looking intersections by remembering features of its previous locations. For example, the robot may know that it should turn right at the upcoming intersection because it remembers that it just passed the elevator doors and because the elevator doors are not next to a left-turn intersection.

Unique infrared transceivers installed at the intersections in the hospital might avoid the need for memory, but this installation may not be possible because of another important perceptual limitation listed above: limited funds. The "hardware solution" that uses many transceivers can be considerably more expensive than a "software solution" that uses short-term memory.

Driving The agent suffered from utile hidden state because its could not see the car in front and the blind spot at the same time.

With short-term memory, the driver would look at the car ahead, remember that it was close enough for passing, look at the blind-spot for a clear way, and once it had both elements of the relevant conjunction, it could make the decision to begin passing.

Note that the short-term memory does not necessarily have to consist of raw perceptions or actions. It could be memory of contingencies ("if I see a gas station, I should pull into it"). It could also be a memory specifying a the location of program counter in a plan ("I'm at step three of my current plan. That implies I've already walked in front of the chair, and I've already turned around; next I should try to sit down"). A paper by Chrisman, Caruana and Carriker [Chrisman *et al.*, 1991] contains a description and comparison of different styles of memory.

Since an agent with memory could execute several perceptual actions in sequence and remember the results from each of them, and because the agent has the opportunity for decision-making before each perceptual action, the agent can choose which parts of the world to sense using a decision tree [Tan and Schlimmer, 1990; Chrisman and Simmons, 1991; McCallum, 1995b]. Such an approach is far more flexible and efficient than requiring that agent to obtain all the sensory data it may need in one shot. This technique demonstrates a classic benefit of selective perception, but it also requires memory.

Here is my point: By proposing the use of short-term memory, I advocate a "middle path" between purely reactive agents, which suffer from hidden state problems, and traditional planning agents, which require complete world models. The agent must represent the world state with more than it current percepts, but the agent does not require a full predictive model of the entire environment.

The idea of an intermediate level of cognition in between high-level planning and low-level neural phenomena is discussed by Ballard, Hayhoe, Pook and Rao in [Ballard *et al.*, 1996]. This intermediate level is termed the *embodiment level.*

"Hidden state and selective perception are intimately related."

In the explanation of the previous two principles, we discussed selective perception and hidden state separately, and the sense in which selective perception can bring on hidden state. Now I explain in further detail an idea introduced in chapter 1: while seemingly opposites, the problems of selective attention ("too much perception") and hidden state ("too little perception") are actually quite similar. The essence of each can be accurately described in terms of the other.

Selective perception can be viewed as "purposefully making hidden state." When the agent's sensors provide the agent with more distinctions than are relevant to its task, the agent can hide the irrelevant state distinctions by ignoring them, and thus "hiding" those state distinctions.

Uncovering hidden state can be seen as a problem of selective perception. In order to uncover non-Markov hidden state, the agent must look backwards in time and add state distinctions based on past features. This results in a problem of feature selection—only now instead of selecting features only from the current percept, the agent selects features from the past percepts also. Uncovering non-Markovian hidden state is a huge selective perception problem.

One can think of perception as a vector of features. The agent's perceptual space has as many dimensions as this vector has elements. When the agent looks backwards into time, it spreads this one dimensional vector out into a second dimension, adding one element to the time dimension for every step backwards in time the agent considers. The agent must then select from this two-dimensional sea of features.

Another way in which selective perception and hidden state are intertwined is that overt selective perception inherently serializes perception. This can be desirable, but as soon as it is serialized, the agent needs memory in order to be able to take advantage of it.

Selective perception and hidden state refer to opposite problems, but they have related solutions.

"Learning to select perceptual features and use memory is difficult and requires special algorithms."

If we endow a learning agent with memory, it must figure out what to remember and what to forget; if we endow a learning agent with selective perception, is must figure out what to select. If the agent does not make enough state distinctions it will not be able to choose the correct actions; if it makes too many distinctions it will use up resources and will lose generalization. Finding the right dependencies between state distinctions and the task is not trivial.

Here are three chief reasons that finding relevant distinctions is difficult. Note that finding memory distinctions is especially difficult.

- Finding dependencies between the agent's state distinctions and the task is difficult because the space of possible distinctions is large. With each feature, the size of the potential state space grows exponentially with respect to the number of features. With every step backwards in time, the number of distinctions to consider grows exponentially again—the size of the state space then grows super-exponentially! Any time the solution consists of a sequence of actions with cumulative effects, the space of possible solutions is exponential in the length of the sequence. In memory-less agents this super-exponential growth of state space is hidden: the size of the space of policies is the number of actions to the power of the number of possible percepts. In essence, with completely observable state, the number of significant world states is bounded by immediate perception. With hidden state, the agent knows no bounds on the number of world states, and the exponential nature of action sequences is unleashed. An agent learning to use memory must not only learn the mapping from states to actions, but must also learn the correct state space.

- A stochastic world may not demonstrate consistent relationships between features and outcomes. Useful features cause distinctions that help predict future circumstances. When a certain distinction is not a perfectly consistent predictor of reward, it doesn't mean it should not be kept—it will just become more difficult to know which distinctions are worth keeping and which are not.

- Exploration of the world state space is made more difficult because, before the agent makes distinctions, it cannot even recognize different parts of the state space as being different; this is especially true of hidden state. Thus, the agent cannot tell whether it has visited a particular state before or not. Choosing when to explore new actions versus when to exploit current knowledge is already a difficult problem without hidden state; the need for memory makes the problem even harder. I call this the "hidden exploration problem." In a separate paper, I discuss this problem further and propose an approach to improving exploration with hidden state [McCallum, 1996].

"Experience is expensive."

Agents should learn in as few trials as possible. Learning experience is expensive in terms of wall clock time, and dangerous in terms of potential damage to the robot and its surroundings. Experience is often relatively more expensive than storage and computation. Furthermore, the expense of computation and storage will only decrease as advances in computing hardware continue; the cost of experience will not.

Gathering experience can easily take several orders of magnitude more time than the necessary computation. For example, Jeff Schneider trained Rochester's robot to throw a ball at a specified target [Schneider, 1994]. After each trial, Jeff had to walk to the ball, pick it up off the floor and put it back in the robot's hand. For the sake of Jeff's time and back muscles, it was more important to reduce the number of trials than it was to reduce the amount of computation or computer storage.

The field of "active learning" studies methods by which the number of experiments in the environment can be economized [Cohn, 1993; Moore and Schneider, 1995; Cohn *et al.*, 1996].

"Agents must be able to handle noisy perceptions and actions."

Agents should be able to handle noisy perceptions and actions because the agent perceives a stochastic world. Sensors are noisy; invisible hidden state looks like noise; and random things really happen in the world.

Building agents that only handle deterministic environments is severely limiting.

"Final performance is to be balanced against training time."

We should aim for reasonably good behavior given the complexity of the task—not necessarily optimal performance. In many cases, it is not worth running an algorithm that will take ten times more trials to learn good performance, in order to gain only a one-twentieth increase in that performance. In a changing environment, this is especially true [Wisniewski and Brown, 1993].

3 Background

Chapter Outline

This chapter presents the technical background for the dissertation. It introduces reinforcement learning, Markov models, and partially observable Markov decision processes. It also discusses the specifics of some related work in reinforcement learning.

3.1 Reinforcement Learning

Reinforcement-learning agents improve their performance on sequential tasks using reward and punishment received from their environment. They are distinguished from supervised learning agents in that they have no "teacher" that tells the agent the correct response to a situation when an agent responds poorly. An agent's only feedback—only indication of its performance on the task at hand—is a scalar reward value (also called credit).

Reinforcement learning is a problem formulation, not a solution technique. The siblings of reinforcement learning are supervised learning and unsupervised learning—*not* neural networks, genetic algorithms, hill climbing, generate-and-test, et cetera, which are all solution techniques that can be applied to problem formulations like reinforcement learning and supervised learning.

The usual approach taken by reinforcement learning agents involves associating expected reward values with different agent internal states, and using those reward values to choose actions.

Reinforcement-learning agents are faced with two credit assignment problems. The first, the *structural credit assignment* problem, involves distributing reward over similar world states. In domains with large state spaces the agent cannot visit every possible state, and thus cannot learn distinct expected reward values for every state. The agent should use some method for assigning credit across the *structure* of the state space so that the reward values in similar states can be used to choose an appropriate action in states the agent has never seen before. This is exactly the generalization problem. Selective perception is a way of combining similar states.

The second problem, the *temporal credit assignment* problem involves distributing reward over the sequence of state-action pairs that lead up to that reward. When beneficial or decisively harmful actions are not rewarded or punished immediately, but some time later, we say that reinforcement is delayed. In this case, the agent must use some scheme for assigning state-action pairs some accumulation of the rewards and punishments that the agent receives after leaving that state.

3.1.1 A Model of the Agent and its Environment

This section provides formal notation for describing the environment and the agent. We will approximate the world by making the state space, the action space and time discrete. At each time step, an action is executed, and the world instantaneously transforms itself from its previous state to the new state that results from the action.

Define $\mathcal{X}$ to be the finite set of discrete states of the world, and define $\mathcal{A}$ to be the finite set of discrete actions. We write the current world state at time t as x_t, where $x_t \in \mathcal{X}$; we write the agent's action as a_t, where $a_t \in \mathcal{A}$.

The transition function T models the results of actions executed from states, and may be stochastic. Let x be a random variable on the set of states of the world, $\mathsf{x} \in \mathcal{X}$. We write

$$T(x_t, a_t) = \mathsf{x}_{t+1}. \tag{3.1}$$

We can also refer to the probability that a certain action, executed in a certain state will result in some particular state.

The probability that executing action a_j in state x_i will result in state x_k is

$$\Pr(T(x_i, a_j) = x_k). \tag{3.2}$$

We will abbreviate this as follows:

$$\Pr(x_k | x_i, a_j). \tag{3.3}$$

The tuple $(\mathcal{X}, \mathcal{A}, T)$ defines the agent's environment.

Now we turn to a formalism for the agent.

Define $\mathcal{R}$ to be a scalar range of possible rewards. We write the agent's reward at time t as r_t, where $r_t \in \mathcal{R}$. The reward function R models reward from the environment, and may be stochastic. Let r be a random variable on the scalar range of possible rewards. We write

$$R(x_t, a_t) = \mathsf{r}_{t+1}. \tag{3.4}$$

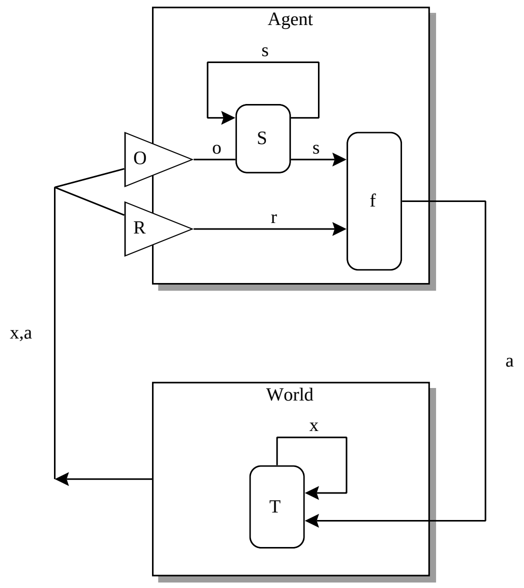

Figure 3.1: A diagram of the interactions between the agent and its world.

Define $\mathcal{O}$ to be the finite set of the agent's observations. We write the agent's current perception o_t, where $o_t \in \mathcal{O}$. The sensory input function O maps from states of the world and actions of the agent to agent perceptions, and may be stochastic. Let o be be a random variable on the set of possible observations, $\mathsf{o} \in \mathcal{O}$. We write

$$O(x_t, a_t) = \mathsf{o}_{t+1} \tag{3.5}$$

Define $\mathcal{S}$ to be a finite set of discrete states of the agent. At time t, we write the agent's current internal state s_t, where $s_t \in \mathcal{S}$. The internal state function maps from internal states at time $t-1$, actions at time $t-1$ and perceptions at time t to internal states at time t. (This is the same as the next-state function of a finite state automaton.)

$$S(s_{t-1}, a_{t-1}, o_t) = s_t \tag{3.6}$$

In the case of reactive agents where the agent's internal state is determined entirely by the current perception, the internal state at the previous time step has no effect and there is a one-to-one correspondence between elements of $\mathcal{O}$ and $\mathcal{S}$.

$$s_t \equiv o_t \tag{3.7}$$

The tuple $(R, \mathcal{O}, O, \mathcal{S}, S)$ defines the interface between the agent and its environment.

If we assume that we have a memory-less agent with no perceptual aliasing (i.e. there is a one-to-one correspondence between the elements of $\mathcal{X}$, $\mathcal{O}$ and $\mathcal{S}$), the agent's internal states follow a finite state Markov decision process. For the rest of this section on reinforcement learning I will write s for both s and x.

3.1.2 Agent Control

The behavior of the agent is defined by a *policy* function that returns the agent's choice of action at each time step. The policy function maps the agent's internal state to the actions it chooses. We call this function f, and write

$$f(s_t) = a_t \tag{3.8}$$

The policy function may also be stochastic.

The goal of a reinforcement learning agent is to modify its policy function in order to maximize some measure of the rewards it receives. We will use a measure comprised of the expected discounted sum of the total reward received over time. This value, called *return*, and written $\mathbf{r}$, is defined at time t as

$$\mathbf{r}_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^n r_{t+n} + \ldots \tag{3.9}$$

where $0 \leq \gamma \leq 1$ is a temporal discount factor that effects how much future rewards are valued at the present. As long as γ is strictly less than one, then the effect of distant future reward becomes negligible, and return is well-defined for tasks whose duration is infinite.

An alternative to using discounting to prevent the infinite-horizon future rewards from diverging is *average reward* [Mahadevan, 1996]. Another alternative is to consider only *finite-horizon* policies, in which the agent calculates a policy that will be optimal given that the agent stops after a fixed, finite number of steps.

Bellman formalized the technique he called *dynamic programming* as a solution to sequential decision problems [Bellman, 1957]. Howard gave extensive treatment to the relation between dynamic programming and Markov chains [Howard, 1960].

3.1.3 Q-learning

One-step Q-learning [Watkins, 1989; Watkins and Dayan, 1992] has attracted much attention as an implementation of reinforcement learning because it works well and because it is derived from dynamic programming [Bellman, 1957].

Dynamic Programming by Value Iteration

The agent can learn the optimal policy by a variation on dynamic programming. We will use a textbook definition of dynamic programming to guide the development of value iteration for agent control [Corman *et al.*, 1990]:

> "Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal value. The development of a dynamic-programming algorithm can be broken into a sequence of four steps.
>
> 1. Characterize the structure of an optimal solution.
> 2. Recursively define the value of an optimal solution.
> 3. Compute the value of an optimal solution in a bottom-up fashion.
> 4. Construct an optimal solution from computed information."

We begin, (1), by characterizing the structure of an optimal solution for the reinforcement learning agent. The goal of the agent is to maximize its expected return from each state s. If the agent begins in state s and follows policy f, the expected return is the value function, $V_f(s)$, defined

$$E[V_f(s)] = \mathsf{r}_1(s,f) + \gamma \mathsf{r}_2(s,f) + \ldots + \gamma^{n-1}\mathsf{r}_n(s,f) + \ldots \tag{3.10}$$

where $\mathsf{r}_n(s,f)$ is a random variable indicating the reward received at time n if the agent begins in state s at time 0 and follows policy f. The optimal solution will be a policy function, f, that maximizes $V_f(s)$ for all $s \in \mathcal{S}$.

Next, (2), we rewrite the value function recursively:

$$E[V_f(s)] = \mathsf{r}_1(s,f) + \gamma V_f[T(s, f(s))] \tag{3.11}$$

Now, (3), we describe a method for computing the optimal solution in a bottom-up fashion. Both *value iteration* and *policy iteration* are bottom-up techniques that can solve the above recurrence relation for $V_f(s)$ [Howard, 1960]. Here we will derive Q-learning using value iteration.

Given that the functions R and T are known, value iteration calculates correct values for $V_f(s)$ by solving optimality equation 3.11 on successively longer finite length tasks. Thus, as with all dynamic programming problems, the solutions to the larger problems are built using the already-calculated solutions to the sub-problems. Define $V_f^n(s)$ to be the expected return if the agent begins in state s and follows policy f for only n steps. The base case, $V_f^1(s)$ is easily obtained directly from the reward function R. For all $s \in \mathcal{S}$ let

$$V_f^1(s) = \max_{a \in \mathcal{A}} R(s,a). \tag{3.12}$$

In the inductive case, we calculate $V_f^{n+1}(s)$ using R and $V_f^n(s)$: For all $s \in \mathcal{S}$ let

$$V_f^{n+1}(s) = \max_{a \in \mathcal{A}} \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} \Pr(s'|s,a)\, V_f^n(s') \right) \tag{3.13}$$

Given $\gamma < 1$, repeated applications of equation 3.13 with increasing n will yield values of $V_f^n(s)$ arbitrarily close to $V_f(s)$. This "iterative" process, repeatedly looping over all transitions (s, a, s') with successively larger n, gives value iteration its name.

Finally, (4), we construct an optimal policy using our computed solutions to $V_f(s)$. So that we can compare the values of different possible actions to be executed from a state, we index these return estimates by initial actions, a, as well as by starting states, s. The new quantity is written $Q(x, a)$, and is called an *action-value*. It is the expected return for taking action a from state s and following policy f thereafter.

$$Q_f(s, a) = R(s, a) + \gamma E[V_f(T(s, a))]. \tag{3.14}$$

This is the expected return for taking action a from state s and following policy f thereafter. $V_f(s)$ is related to $Q_f(s, a)$ by

$$V_f(s) = \max_{a \in \mathcal{A}} Q_f(s, a). \tag{3.15}$$

We can define an optimal policy in terms of these values. From any state, the agent would choose the action that is associated with the greatest expected return:

$$f(s) = \mathrm{argmax}_{a \in \mathcal{A}} Q(s, a). \tag{3.16}$$

Reinforcement Learning and Q-values

The agent will store approximations to $V(s)$ and $Q(s, a)$. The agent's estimate of $V(s)$ is usually written $U(s)$ (for "utility"); the agent's estimate of $Q(s, a)$ is usually written again $Q(s, a)$. A reinforcement learning agent cannot calculate correct values for these approximations by value iteration directly, because it does not know the reward function R, or the transition function T *a priori*. The agent updates $U(s)$ and $Q(s, a)$ incrementally, but instead of following a systematic iteration over all transitions (an operation that depends on R and T), it performs a "world-directed" iteration, updating an estimate only when it experiences a transition and a reward in the world.

At each step, when the agent makes the transition from state s_{t-1} to state s_t by executing action a_{t-1}, the agent adjusts a Q-value with the rule

$$Q_f(s_{t-1}, a_{t-1}) \leftarrow (1 - \alpha)\ Q_f(s_{t-1}, a_{t-1})\ +\ \alpha\ (r_t + \gamma U(s_t)). \tag{3.17}$$

The agent updates $Q_f(s_{t-1}, a_{t-1})$ with a weighted average of its previous value and the new expected reward information from the next iteration on this transition. The last parenthesized part of equation 3.17 corresponds to applying equation 3.13 with a particular transition and thus a fixed s, a, s'. The weighting in the average, called the *learning rate*, is α, where $0 \leq \alpha \leq 1$.

The agent then updates the utility of state s_{t-1} by

$$U_f(s) \leftarrow \max_{a \in \mathcal{A}} Q_f(s, a). \tag{3.18}$$

As shown above, the agent can define its policy in terms of its Q-values. The agent's deterministic policy f is

$$f(s_t) = \mathrm{argmax}_{a \in \mathcal{A}} Q(s_t, a). \tag{3.19}$$

For convergence, the learning rate must decay and there must be sufficient exploration. Some stochasticity can accompany the agent's policy in order to cause exploration of different actions. A simple randomization strategy policy is to output an action chosen uniformly at random with some probability, and otherwise output the Q-value specified action.

Watkins and Dayan have shown that under the appropriate conditions, Q-learning is guaranteed to converge to the optimal policy [Watkins and Dayan, 1992].

An alternative to Q-learning is to learn T and R, and then do value iteration directly on the learned model [Barto *et al.*, 1995]. The approach is particularly desirable when the state space is simple enough that it can be modeled well, and when making steps in the world expensive. Some other techniques fall mid-way between this approach and Q-learning: DYNA [Sutton, 1990b], Q-DYNA [Peng and Williams, 1992], and Prioritized Sweeping [Moore and Atkeson, 1993].

3.1.4 Hidden State

If we remove the assumption that there is a one-to-one correspondence between the agent's observations and states of the world, (*i.e.* discard equation 3.7), the world state and observation are again related only by the O function, and then the agent is said to have *partial observability* of its environment. The agent's interaction with this environment is described by a *partially observable Markov decision process* (POMDP) [Astrom, 1965].

Partially Observable Markov Decision Processes

A partially observable Markov decision process is comprised of a finite set of states, $\mathcal{S} = \{s_1, s_2, ..., s_N\}$ and a finite number of observations (percepts), $\mathcal{O} = \{o_1, o_2, ..., o_M\}$ and a finite set of actions $\mathcal{A} = \{a_1, a_2, ..., a_K\}$. For each state there is a vector of observation probabilities; we write $\Pr(o_i|s_j)$ for the probability of seeing observation o_i while in state s_j. For each state-action pair there is a vector of transition probabilities; the notation $\Pr(s_k|s_i, a_j)$ signifies the probability that executing action a_j from state s_i will result in state s_k. The agent's belief about the state of the world is maintained in a vector of its state occupation probabilities, written $\vec{\pi}(t) = \langle \pi_1(t), \pi_2(t), ..., \pi_N(t) \rangle$, where $\pi_i(t)$ is the agent's belief that the world state is represented by s_i at time t.

To calculate the state occupation probabilities at time $t+1$, the agent uses Bayes' rule (where k is whatever constant is needed to make $\sum_i \pi_i(t+1) = 1$):

$$\pi_i(t+1) = k \cdot \Pr(o_{t+1}|s_i) \sum_j \Pr(s_i|s_j, a_t)\, \pi_j(t). \tag{3.20}$$

In a partially observable environment, immediate observations may not provide enough information for the agent to uniquely identify the current world state, and thus choose the best action. Some form of memory or context is necessary. There are many structures that could be used to hold memory. One powerful choice, if the identity of all relevant world states and the transition function is known, is for the agent to represent directly this full POMDP of world states, and use the vector of POMDP-state occupation probabilities as its internal state. This vector is then called the agent's *belief state*. Unfortunately, using the belief state as the domain for the policy is unwieldy because belief state is high-dimensional and continuous, (having as many dimensions as there are states in the world, and each dimension taking on the value of that state's occupation probability). Work that addresses this difficulty is discussed in section 3.2.

Other methods for representing memory is also discussed in section 3.2.

Introductions to POMDP's can be found in [Lovejoy, 1991] and [Bertsekas, 1976]. Since POMDP's can be viewed as hidden Markov models with actions added, readers may also benefit from the introductory material on hidden Markov models in [Rabiner, 1989] and [Poritz, 1988].

If we want to use traditional POMDP solution techniques for reinforcement learning, we will have to learn the model. The obvious strategy is to use Baum-Welch-like methods. Chapter 4 discusses one such approach.

3.2 Reinforcement Learning with Hidden State

This section discusses some related work in reinforcement learning with hidden state.

3.2.1 Stateless Agents

Some approaches to the hidden state problem do not attempt to disambiguate the aliased states.

The simplest approach is simply to ignore the hidden state problem and apply traditional reinforcement learning methods as if there were no aliased states. Some experience has shown that in certain non-Markovian environments this approach can work [Barto *et al.*, 1983], however, there are many cases in which ignoring hidden state results in complete failure; several papers give examples with explanations [Whitehead, 1992; Singh *et al.*, 1994; McCallum, 1993b].

A more careful solution to the hidden state problem is to avoid passing through the perceptually aliased states. This is the approach taken in Whitehead's Lion algorithm [Whitehead, 1992]. Whenever the agent finds a state that delivers inconsistent reward, it sets that state's utility so low that the policy will never visit it again. The success of this algorithm depends on a deterministic world and on the existence of a path to the goal that consists of only unaliased states.

Other solutions do not avoid aliased states, but do the best they can given a non-Markovian state representation while also evading the disasters that would result from ignoring the hidden state altogether [Littman, 1994a; Singh *et al.*, 1994; Jaakkola *et al.*, 1995]. They involve either learning deterministic policies that execute incorrect actions in some aliased states, or learning stochastic policies that, in aliased states, may execute incorrect actions with some probability. These approaches do not depend on a path of unaliased states, but they have other limitations: when faced with many aliased states requiring different actions, the probability of executing an optimal sequence of actions falls precipitously; when faced with potentially harmful results from incorrect actions, deterministically incorrect or probabilistically incorrect action choice may prove too dangerous; and when faced with performance-critical tasks, inefficiency that is proportional to the amount of aliasing may be unacceptable.

3.2.2 Agents with Memory

The most robust solution to the hidden state problem is to use memory to disambiguate the aliased states. *State identification* techniques uncover the hidden state information—that is, they make the agent's internal state space Markovian. This transformation from an imperfect state information model to a perfect state information model has been formalized in the decision and control literature [Bertsekas and Shreve, 1978]. By augmenting the agent's percepts with history information—short-term memory of past perceptions, actions and rewards—the agent can distinguish perceptually aliased states, and can then reliably choose correct actions from them.

The example tasks in chapter 2 illustrate the use of memory to disambiguate hidden state: in the examples, the robot knows which replacement part to pick because it remembers seeing the faulty part; the robot knows whether to turn right or left at the intersection because it remembers having passed the elevator door; the robot knows to pull into the passing lane because it remembers having looked both in front and behind. Using memory to disambiguate the aliased situations allows the robot to efficiently perform the task independent of the number of aliased states in that path. It allows the robot to choose correct actions reliably, as opposed to making probabilistic guesses whenever faced with ambiguous sensations.

Several research efforts have been devoted to learning a policy in cases where the agent is provided with a state model of its environment.

When the agent is provided with a POMDP model of its environment and the agent uses *belief state* as the state space on which to specify its policy, the key difficulty is that this state space is continuous and has as many dimensions as there are POMDP states.

However, Smallwood and Sondik have shown that the optimal value function for any POMDP can be approximated arbitrarily well by a piecewise-linear and convex function [Smallwood and Sondik, 1973], and a certain class of POMDP's have value functions that can represent exactly with such functions [Sondik, 1973]. Littman, Cassandra and Kaelbling have presented algorithms for computing these functions when given the POMDP [Littman, 1994b; Cassandra *et al.*, 1994; Littman *et al.*, 1995]. The last of these papers presents empirical results on a POMDP's with 57 and 89 states.

Bacchus, Boutilier and Grove [Bacchus *et al.*, 1996] describe an algorithm that, given a state transition function, reward function and world state model that is Markovian with respect to state transitions, but not Markovian with respect to reward, will augment the given model in order to provide Markovian rewards. The algorithm expands the provided state space by describing the non-Markovian aspects of the system using formulas of temporal logic, termed PLTL. Once the state space is expanded, traditional methods such as Value Iteration can be used to create a policy. They note that in some cases the search for distinguishing formulas can be complex—exponential in the size of reward formulas.

In a later paper, Bacchus, Boutilier and Grove [Bacchus *et al.*, 1997] describe a method that uses the same PLTL temporal logic, but rather than creating an explicit set of states, to which one applies Value Iteration, they instead create a "structured" state representation. In a structured representation, different world states are represented by different combinations of value assignments to state variables; these have the potential to be much more compact than an explicit enumeration of all states. The policy is then calculated directly in the structured representation. There are no empirical results in either of the above two papers.

Structured representations are also used in another approach, based on Bayesian Networks. Boutilier and Poole [Boutilier and Poole, 1996] take a provided POMDP model of the world with Markovian reward, and use Bayesian Networks with tree-shaped Structured Conditional Probability Matrices [Boutilier *et al.*, 1996] to build a compact representation of belief space. Unfortunately, the algorithm only works for finite-horizon policies and necessitates building a number of trees exponential in the time-horizon of the agent. There are no empirical results.

In a related approach, Boutilier, Dearden and Goldszmidt [Boutilier *et al.*, 1995] use trees to construct an approximately optimal policy when the action-model and reward function are known, but instead of using a structured representation to capture hidden state, they use the structured approach to allow for state generalization. Again, they build separate trees for each time step of the finite-horizon task. Boutilier and Dearden [Boutilier and Dearden, 1997] extend this work by pruning the trees—resulting in approximate solutions, but smaller trees.

Other solutions do not require that the agent be provided with a state model, and use memory, but the memory representation is predefined and fixed.

For example, Lin and Mitchell present results with agents that use a fixed-size finite-window of past percepts and actions [Lin and Mitchell, 1992][1].

ALECSYS [Dorigo, 1992] is a learning classifier system implemented with a genetic algorithm [Holland, 1975; Booker *et al.*, 1989]. In work by Dorigo and Colombetti [Colombetti and Dorigo, 1994], ALECSYS learns proper sequences by maintaining internal state indicating which subtask the agent should be performing. Transition signals produced by the trainer or environment prompt the agent to switch from one subtask to another. In other work by the same authors [Dorigo and Colombetti, 1994], ALECSYS is modified to include among its percepts indications of whether a sensor value changed in the last N steps, where N is a parameter of the algorithm set by the robot designer. This is a variation of the fixed-sized perception window approach to memory, with the

[1]Results with recurrent neural networks are presented in the same paper. See page 37

simplification that the agent remembers only the value since the last change, not all sensor values for the last N steps.

Predefined, fixed memory representations such as constant-sized perception windows (more formally known as order-n Markov models) are often undesirable. When the length of the window is more than needed, they uselessly increase the number of agent internal states for which a policy must be stored and learned by an exponential amount; when the length of the memory is less than needed, the agent reverts to the disadvantages of undistinguished hidden state. Even if the agent designer understands the task well enough to know its maximal memory requirements, the agent is at a disadvantage with constant-sized windows because, for most tasks, different amounts of memory are needed at different steps of the task.

3.2.3 Agents that Learn a Model and Learn to Use Memory

A more flexible approach is to learn the agent's memory structure on-line. The issue then becomes how to represent this memory and how the agent should learn what to remember and what to forget. Because the agent begins without knowing how to perform the task at hand, the agent also begins without knowing how much memory will be required.

Let us emphasize the way in which learning how much to remember is a departure from much other work in machine learning for multi-step tasks. Memory-learning agents must not only learn a mapping from states to actions (which is often difficult enough as it is), but learn the required *state space* as well. When confronted with hidden state, the agent can no longer rely on perception to completely define the agent's internal state space. The internal states of the agent must depend on both the current perception and some variable, learned amount of memory about past perceptions and actions.

3.2.4 Related Work in Agents that Learn to Use Memory

Several reinforcement learning algorithms are based on techniques that adapt the memory representation on-line. This subsection provides a survey of them.

Finite History Window Traces

One simple way to add memory to the agent's state representation it to have it define its state relative to a finite-sized window of the current and past observations and actions.

The work of Lin and Mitchell [Lin and Mitchell, 1992] and Dorigo and Colombetti [Dorigo and Colombetti, 1994], mentioned above, both use variations on finite-sized history windows—however, the size of window is not changed dynamically during learning.

The Utile Suffix Memory algorithm presented in chapter 6 uses a variable-length finite-size window, in which the length of the window is changed dynamically during learning. The window can also have differing length in different parts of state space.

Tan's Cost Sensitive Reinforcement Learning [Tan, 1991] also has memory, in that the robot remembers the results of several perceptual actions to define its internal state. In another sense, however, the agent does not have memory, because the agent has no mechanism for maintaining state information from one an overt action to the next. This limitation would prevent the agent from solving any of the hidden state example given in chapter 2. The limitation also makes the agent's task much simpler—the agent no longer has to determine what to remember and when to forget: it remembers the results of all its perceptual actions, then throws all its memory away when it makes an overt action. Cost Sensitive Reinforcement Learning also has the disadvantage that it only handles deterministic environments.

Finite State Machines

An agent can represent memory by using a finite state machine (such as a partially observable Markov decision process) as an internal model of its environment. Because the new state occupation probabilities of the model depend not only on the current percept, but also on the previous state probabilities and the previous action, the use of a finite state automaton can provide an agent with context—memory of past observations and actions. For example, the transition probabilities in the model may be set such that it is extremely unlikely that the agent has arrived in a particular state unless the agent has come from some other particular previous state. By building chains of such states with exclusive transition probabilities, the occupation of a state can represent an arbitrary amount of memory about past percepts and actions.

The finite state machine approaches have the advantage that there is much established formal study of finite state machines, hidden Markov models and partially observable Markov decision processes. They are also more expressive than finite sized windows in that they can represent the concept of a state-transition loop in the environment. However, they do not necessary represent the environment as efficiently as some other alternatives. For instance, if the agent needs to remember only one bit of information, but must remember that bit for a long time, the entire state space covered during the intervening time must be duplicated; recurrent neural networks or register-based memories could represent this memory more efficiently.

The Perceptual Distinctions Approach (PDA) [Chrisman, 1992a] and Utile Distinction Memory(UDM) [McCallum, 1993b] both learn POMDP's via state-splitting strategies. These algorithm will be discussed more in chapter 4.[2] The major difference between PDA and UDM is that UDM only splits states when the statistics show that doing so will help predict reward; PDA splits states in order to predict observations.

Wiering and Schmidhuber [Wiering and Schmidhuber, 1996] use Levin Search [Levin, 1973] to search directly in policy space. They introduce Adaptive Levin Search, which

[2]Although this subsection is about techniques that build memory representations, I should point out that other research has studied how to calculate an optimal policy *given* a model (often in the form of a partially observable Markov decision process) [Littman, 1994b; Cassandra *et al.*, 1994; Littman *et al.*, 1995]. PDA and UDM, on the other hand, both emphasize the learning of the finite state machine itself, then use a simple heuristic for calculating the policy from it.

uses experience to increase probabilities of instructions occurring in programs found by Levin Search. They demonstrate their algorithm solving a 39x38 maze with much hidden state in 97 million steps.

The environment learning approach implemented in *Toto* [Mataric, 1990] also learns a finite state machine, and is impressively fast—it learns from single presentations. However, the technique is designed specifically for mobile robots inside rooms and corridors. Unlike the above four algorithms it depends integrally on dead reckoning to disambiguate states, and thus is tied to navigation. In navigational tasks, the global state is directly correlated with geographical position, which is exactly what dead reckoning provides. A meaningful analogy to global state calculating "dead reckoning" for general, non-navigational tasks is not apparent.

I should also mention some related work that, while not associated with reinforcement learning, does provide other examples of learning finite state machines.

The map learning technique described in [Dean *et al.*, 1992] learns finite state models of the environment. It is not applicable to hidden state because it requires a unique, unaliased landmark for each world state, (although the perception of the landmarks is allowed to be noisy); also it does not handle stochastically unreliable actions.

The diversity-based inference procedure for finite automata [Rivest and Schapire, 1987a; Rivest and Schapire, 1987b] learns an ingenious representation called an *update graph* that can efficiently capture regularities in the structure of the environment. The diversity approach finds a set of canonical tests using repeatable cycles in the environment. The Rivest-Schapire algorithm depends on a completely deterministic environment. Work on a neural network implementation of this approach can accommodate noisy sensations, but has other disadvantages [Mozer and Bachrach, 1990].

Hidden Markov models (HMM's) have been used to represent the agent's context in many different kinds of tasks. Rimey used HMM's to choose sequences of sensing actions [Rimey and Brown, 1991; Rimey, 1993]. Pook and Ballard used HMM's to represent context in sequences of manipulative actions with the MIT/Utah hand [Pook and Ballard, 1992; Pook, 1995].

Recurrent Neural Networks

Several approaches have used recurrent neural networks to maintain memory. Backward looping connections from hidden units or output units feed into the input layer—thus the current inputs are partially determined by the networks' own context units, which in turn were determined by previous inputs. Repeated looping of activation through these recurrent connections can maintain memory over several steps.

Schmidhuber [Schmidhuber, 1991] uses two interacting fully recurrent networks—one to predict the environment, and another to choose actions. The networks are trained with the IID-Algorithm [Robinson and Fallside, 1987].

The Recurrent-Q algorithm [Lin, 1993] trains a recurrent neural network containing a fixed number of hidden units, with a certain subset designated as recurrently-connected context units.

Meeden, McGraw and Blank [Meeden *et al.*, 1993] also use recurrent networks for reinforcement learning.

An advantage of using recurrent neural networks for memory is that their representational efficiency can be greater than finite state machines—they can more efficiently memorize one bit over many steps, for instance.[3] They also have the potential to handle continuous inputs and outputs.

One disadvantage of recurrent neural networks is that, unlike finite state machines with state splitting rules, the memory capacity is fixed before learning begins since the number of hidden units is fixed.

The algorithms also take very many steps to learn. Neural networks in general are not well known for quick convergence, and the convergence of recurrent neural networks is typically slower than feed forward networks.

All gradient descent algorithms have problems with local minima. The networks seem to converge to good solutions when applied to simple problems, but they tend to fall into local optima when faced with more complex problems.

Hierarchical Neural Networks

Mark Ring has developed a reinforcement learning agent that uses a hierarchical neural network to solve tasks with hidden state [Ring, 1994]. The algorithm learns on-line, handles stochastic environments, and takes advantage of its hierarchical structure to learn complex behaviors by combining more primitive ones.

It is based on a new kind of neural network he terms Temporal Transition Hierarchies. They are single-layer networks with linear activation functions—however they overcome the usual weakness and represent non-linear classifications by incorporating "high-level units" whose role it is to dynamically modify the weights in the rest of the network. The network also contains units for representing perceptual primitives and for representing actions.

These high-level units are created and added to the network on-line, whenever unit activations are unreliably predicted by the network's connection weights. More specifically, the algorithm looks at the average weight change and average magnitude of the weight change during learning, and when it sees that the weights are being pulled in two different directions, it creates a new high-level unit. A high-level unit holds context over time in that its activation can effect the activation of an action unit at the next time step.

The algorithm is related to UDM [McCallum, 1993b], the Perceptual Distinctions Approach (PDA) [Chrisman, 1992a] and USM [McCallum, 1995b] (which will be discussed in chapter 6), in that Temporal Transition Hierarchies looks progressively backward in time and look at statistical differences in order to decide when to create new representational power. However, its new statistical test for such additions is subject to more

[3]While it can be argued that recurrent neural networks *are* finite state machines unless the weights can have infinite precision, in practice, the above statement still holds.

environment-specific parameter tuning than the established Student-T, Chi-Squared and Kolmogorov-Smirnov statistical tests used by UDM, PDA and USM.

Furthermore, the other three algorithms test specifically for a violation of the Markov property by asking a statistical question about two data-sets separated by a time-dependency; thus they can separate (1) variance due non-Markovian dependences, from (2) variance due to a stochastic environment. Temporal Transition Hierarchies, on the other hand, use a test that is independent of the new temporal distinction that will be introduced, and thus may create unnecessary high-level units that will try in vain to predict environmental stochasticity.

Both USM and Temporal Transition Hierarchies use linear history traces to represent context, and thus, unlike UDM and PDA, they assume an order-k Markov environment.

Registers

Genetic Programming with Indexed Memory [Teller, 1994] uses genetic programming [Koza, 1992] to evolve programs that apply load and store instructions to a fixed-sized register bank. These registers, which hold values for later retrieval, obviously supply the agent with memory.

The chief advantage of Genetic Programming with Indexed Memory is its great representational power. In fact, Teller makes a point of emphasizing that the class of potentially evolved programs is Turing complete [Teller, 1993].

Unfortunately, this advantage also causes the algorithm's worst disadvantage—it pays dearly for this flexibility in terms of learning steps required. Genetic Programming with Indexed Memory can take an extremely large number of steps to learn. For example, in Teller's demonstration of the algorithm on a grid world box-pushing task, the population contains 800 individuals, the population must be evolved over 100 generations, after each generation all the individuals must be evaluated on 40 test cases that last for 100 steps each—for a total of over 3×10^8 steps.

Cliff and Ross have applied a classifier system to learning with one- and two-bit registers [Cliff and Ross, 1995]. They find that their approach worked successfully on small problems, but is unlikely to scale well.

3.3 Reinforcement Learning with Selective Perception

This section discusses related work in which a reinforcement learning agent uses selective perception. Several pieces of related work in reinforcement learning have presented algorithms that learn to select which perceptual features will be used to define the agent's state space.

3.3.1 Overt Selective Perception

Some techniques use overt selective perception, in which the agent's policy chooses perceptual actions that direct sensors towards particular features of the environment.

Tan's Cost Sensitive Reinforcement Learning [Tan, 1991] chooses a series of perceptual actions to gain knowledge about features of the world. The perceptual actions have costs associated with them, just like manipulative actions, and the agent learns to balance the benefit of knowledge about world features against the cost of executing the actions to sense them.

Whitehead's Lion Algorithm [Whitehead, 1992] learns to execute redirect visual sensor using visual routines [Ullman, 1984; Agre and Chapman, 1987]. The agent overcomes perceptual aliasing by detecting and avoiding sensor redirections that are inconsistent.

Both Cost Sensitive Reinforcement Learning and the Lion Algorithm are discussed in more detail in section 3.2.

3.3.2 Covert Selective Perception

Other techniques use covert selective perception, in which the agent makes state distinctions based only on a subset of all available features, but the agent's policy does not explicitly choose actions that direct this attention.

Chapman and Kaelbling's G-algorithm [Chapman and Kaelbling, 1991] builds a decision tree by selecting those binary perceptual bits that are correlated with differences in Q-values. The G-algorithm is discussed further in section 7.5.3.

Moore's Parti-game [Moore, 1993] also builds a tree, but this tree builds an adaptive resolution model of a multi-dimensional *continuous* space. Parti-game is discussed further in section 7.5.2.

The U-Tree algorithm, described in chapter 7, is demonstrated on a task in which it uses both overt selective perception and covert selective perception simultaneously.

It could be argued that all forms of value function approximation [Boyan *et al.*, 1995] are examples of covert attention, in that function approximation does not capture the full resolution of the value function—thus attending to certain state distinctions, but ignoring others.

The idea of creating generalization by ignoring certain input features is prevalent in the statistics in supervised learning literature.

3.3.3 Modularity

Various modular reinforcement learning algorithms can also be viewed as using covert selective perception, in that the modular nature of the algorithm causes the agent to attend to some environmental features and ignore others.

Karlsson's Task Decomposition [Karlsson, 1994; Tenenberg *et al.*, 1993; Whitehead *et al.*, 1993] and Humphreys' W-learning [Humphreys, 1996a; Humphreys, 1996b], for example, divide the agent's policy into modules, each of which is designated to perform a sub-task. Each module is given its own reward function and its own subset of the perceptual features.

Wixon's Modular Reinforcement Learning also divides perception among a hierarchically-organized set of modules [Wixson, 1991].

4 Utile Distinctions

Chapter Outline

This chapter discusses the mechanics of making utile distinctions, and presents an example environment that shows the distinctions necessary for calculating the optimal policy.

It introduces Utile Distinction Memory, an algorithm that makes task-relevant state distinctions by splitting states of a partially observable Markov decision process, and tunes the parameters of the model using the Baum-Welch procedure. The algorithm successfully makes utile distinctions while learning to navigate in an 'M'-shaped maze with hidden state. It works both with and without noise.

With this chapter, we begin a series of four chapters, each of which presents a new algorithm. This first chapter focuses on the idea of a litmus test for finding task-relevant distinctions.

The algorithm presented here is closely tied to the formal state model introduced in the previous chapter—the agent uses a partially observable Markov decision process (POMDP) to represent its internal state distinctions, and it uses a variation on a standard technique, Baum-Welch, to learn the parameters of the model.

In a departure from standard techniques, the algorithm adds new distinctions to the model by splitting states of the POMDP during learning. The novel aspect of this algorithm is the test it uses to determine when a distinction is relevant to the task. Unlike some previous algorithms that add new state distinctions whenever they are needed to predict future percepts [Chrisman, 1992a], Utile Suffix Memory only adds new state distinctions whenever needed to predict future reward. This difference causes a fundamental shift in the agent's representational approach. Whereas utility-based distinctions will build an agent internal state space whose size is dependent on the task at hand, perception-based distinctions will build a state space large enough to represent all the world as the agent perceives it.

If the agent's task is simple, but the world is complex, (and the agent can perceive manifestations of the world complexity), we would still like the agent's internal state space to be simple. Ideally, the difficulty of learning should be proportional to the difficulty of the task, not the complexity of the entire perceivable world.

This chapter presents a utile distinction rule, and then an algorithm that uses the rule to add short-term memory distinctions. While this chapter does not discuss distinctions based on selective perception, much of what is said in this chapter also applies to utile distinctions for selective perception. Chapter 7 presents an algorithm that implements selective perception.

4.1 The Key Idea: Utility-Based Distinctions For Memory

How can the agent know whether adding a new memory distinction will help it predict utility?

Often a perceptually aliased state that affects utility will have wildly fluctuating reward values, however, we cannot base a state splitting test solely on reward variance—some changes in reward are caused by the stochastic nature of the world, and adding a distinction will not help the agent more consistently get high reward. The agent must be able to distinguish between fluctuations in reward caused by a stochastic world, and fluctuations that could be better predicted after adding a memory distinction.

Consider what new information the agent gains by splitting a state and dividing the incoming transitions among the resulting states: after splitting a state, the agent has the ability to distinguish between two previously aliased states based on knowledge about which state the agent came from.

The key idea in this chapter is that the agent can use statistical tests based on future discounted reward to find state distinctions that help predict reward. These tests compare reward statistics associated with different incoming transitions to a state. Recall the definition of "Markov with respect to reward": a state is Markov with respect to reward if knowledge of past states does not help predict future reward from that state. Thus, if the state satisfies this Markov property, we would expect the separated reward statistics to be the same. If, on the other hand, there is a statistically significant difference in future discounted reward between the statistics depending on where the agent came from, then knowledge of which state the agent came from does help predict reward, and the state should be split. This comparison of statistics that have been separated according to previous state is exactly a test for a violation of the "Markov with respect to reward" property.

Before describing the algorithm in section 4.3, I discuss the nature of task-relevant distinctions, and present an example that sheds some light on the state distinctions necessary for calculating the optimal policy.

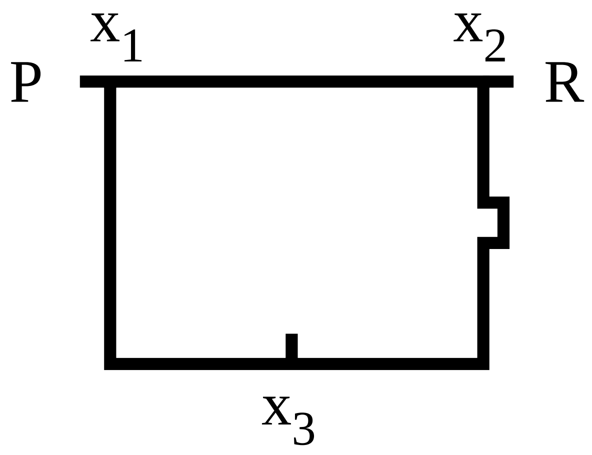

Figure 4.1: An example maze, showing that the state distinctions necessary for representing the optimal policy are not necessarily sufficient for learning the optimal policy. The bend in the path from x_3 to x_2 causes that path to be a little longer than the path between x_3 and x_2.

4.2 The Distinctions Necessary for Learning

Given that the agent explicitly chooses which state distinctions to make, what distinctions are necessary for the agent to calculate the optimal policy? This section presents a simple, yet perhaps, counter-intuitive proof that provides negative results on this question. The example grew out of discussions with Kaelbling [1995] .

Theorem 1 *The state distinctions necessary for representing the optimal policy are not necessarily sufficient for learning the optimal policy.*

The proof is by example. I describe an environment and task for which we can calculate the optimal policy and find a minimum set of state distinctions adequate for representing that policy. Yet, when that small set of state distinctions is used to recalculate a policy in the new reduced state space, we obtain a non-optimal policy.

There are many simple finite state machines that could be used as examples; in fact, the first example shown to me by Kaelbling [1994] was a finite state machine. However, I find the following maze-based example easier to understand because it is more geometrical.

The environment is depicted in figure 4.1. It consists of several connected corridors with three T-shaped intersections. We label the intersections x_1, x_2 and x_3. Intersections x_1 and x_2 have corridors to the east, south and west; intersection x_3 has corridors to the east, north and west.

At each of the T-intersection choice points, the agent has two actions, one for choosing left, one for right. For simplicity, the actions are deterministic. Once an action is executed, the agent follows the corridor, going straight whenever it can, and following the contours of the hallway whenever there is no intersection. Whenever the agent arrives at a dead end, it is instantaneously teleported back to intersection x_3, and positioned facing south. Thus, the agent's only choice points occur when the agent arrives at one of the three T-intersections *and* it is facing the wall of the T. According to the action and time discretization of the reinforcement learner, these three choice points are the three states of the environment—the agent has no action choice at any other locations in maze. The agent's state transition matrix appears in table 4.1

Action "left"

	x_1	x_2	x_3
x_1	0	0	1
x_2	0	0	1
x_3	0	1	0

Action "right"

	x_1	x_2	x_3
x_1	0	0	1
x_2	0	0	1
x_3	1	0	0

Table 4.1: The agent's state transition matrix.

Immediate reward is the sum of two components: some amount of negative reward for every meter of hallway traveled since the last choice point intersection, plus whatever reward is delivered by passing through the special reward or punishment locations in the two teleport-causing dead-ends at the top. The agent's instantaneous rewards are deterministic and appear in table 4.2.

	right	left
x_1	+0.7	-1.0
x_2	+1.0	-1.3
x_3	-0.5	-0.7

Table 4.2: The agent's immediate reward function.

Executing "right" from intersection x_3 sends the agent to intersection x_1; executing "left" from intersection x_3 sends the agent to intersection x_2, but with a little more distance traveled, and thus a little less reward.

Performing dynamic programming on this Markov decision process, using a temporal discount factor of 0.9, results in the Q-table shown in table 4.3.

Thus, in order to act optimally, the agent should execute the actions of the policy shown in table 4.4.

Now let us consider some possible state distinctions the agent could make with sensors.

If, for example, we give our agent a single sensor—a compass that detects whether the agent is facing north or not—then the agent can represent the optimal policy. When

	right	left
x_1	1.64	-0.06
x_2	1.94	-0.36
x_3	0.96	1.03

Table 4.3: The agent's Q-table, calculated from the model without aliasing.

World state	Policy action
x_1	right
x_2	right
x_3	left

Table 4.4: The optimal policy on the world state space.

the sensor indicates "north" (in states x_1 and x_2), the agent turns right; when the sensor indicates "not north" (in state x_3), the agent turns left. These distinctions, states and policy actions are shown in table 4.5.

Agent internal state	World states	Policy action
North	x_1, x_2	right
Not North	x_3	left

Table 4.5: The optimal policy represented using the "north" state distinction.

Although these state distinctions are sufficient for *representing* the optimal policy they are not sufficient for *calculating* the optimal policy.

If the agent uses these state distinctions to build a Markov decision process based on this single distinction, *i.e.* with states x_1 and x_2 aliased, the agent will see the transition and reward functions shown in tables 4.6 and 4.7, respectively.

The new reward values from the aliased state "x_1 & x_2" were obtained by averaging the reward values from the two component states, giving the reward values from each state equal weight. Thus, the resulting values thus depend on the agent having arrived at these estimates by uniform exploration of both actions from state x_3. If the agent obtains reward estimates via some other exploration strategy, the resulting estimates will be different than those shown in table 4.7, but these changes would not effect the outcome of the new optimal policy, because: (1) action "right" will always have a higher utility than action "left," no matter how the two states are averaged, and (2) the policy action from state x_3 depends on the immediate rewards from state x_3, not the utility of the single, aliased destination state.

Action "left"

	x_1 & x_2	x_3
x_1 & x_2	0	1
x_3	0	1

Action "right"

	x_1 & x_2	x_3
x_1 & x_2	1	0
x_3	1	0

Table 4.6: The transition matrix, as experienced by the agent when states x_1 and x_2 are aliased.

	right	left
x_1 & x_2	+0.85	-1.15
x_3	-0.50	-0.70

Table 4.7: The immediate reward function, as experienced by the agent when states x_1 and x_2 are aliased.

Performing dynamic programming on this new, aliased Markov decision process, using a temporal discount factor of 0.9, results in the Q-table shown in table 4.8.

	right	left
x_1 & x_2	2.08	0.08
x_3	1.38	1.18

Table 4.8: The agent's Q-table, calculated from a model with aliasing distinction.

Thus, according to this new Q-table, calculated from the aliased state distinctions, the agent can act optimally by executing the policy shown in table 4.9. However, in state x_3, this policy differs from the optimal policy.

As shown in table 4.4, the optimal policy can be represented using only the "north" state distinction, however, when the agent uses only the "north" state distinction to calculate a policy, the resulting policy is different than the optimal policy.

Thus, we have proved by example that the state distinctions sufficient for representing the optimal policy are not necessarily sufficient for calculating the optimal policy.

However, if the agent adds a second sensor—for example, a left facing sonar that detects whether or not there is a dead-end within a few meters—then the agent can distinguish between all of the three states, and can calculate the optimal policy.

4.2.1 How This Happened

What are the key features that bring about this result?

	World states	Policy action
North	x_1, x_2	right
Not North	x_3	right

Table 4.9: The non-optimal policy calculated using the Markov decision process based on the "north" state distinction.

- There are (at least) two aliased states, (*i.e.*, x_1 and x_2).

  If, on the other hand, there had been no state aliasing, the agent would have all the parameters of the Markov decision process and would be able to calculate the optimal policy.

- The two aliased states have different utility, (*i.e.*, $U(x_1) > U(x_2)$).

  If, on the other hand, the two states had the same utility, the optimal policy would be indifferent to which one the agent arrives at.

- There is slightly lower reward associated with the path to the aliased state that has higher utility, (*i.e.* x_1). That is, ($r_1 < r_2$), where r_1 is the cost of the path to the higher utility state, x_1; and r_2 is the cost of the path to the lower utility state, x_2. "Slightly" means that even though $r_1 < r_2$, it is still the case that

  $$r_1 + \gamma U(x_1) > r_2 + \gamma U(x_2). \tag{4.1}$$

  Because states x_1 and x_2 are aliased, the optimal policy calculations act as if it does not matter which state the agent gets to—and thus, the calculations choose the path with the slightly lower cost. Unfortunately for the agent, this path actually leads to the aliased state with the comparably much lower utility.

  The "slightly" condition ensures that difference in immediate reward isn't so drastic as to cause the optimal policy to select the path to the lower utility state.

- Note that the two aliased states should have the same optimal policy action. Otherwise, the agent would need to distinguish between them in order to represent the optimal policy.

The combination of all these conditions does not seem overly artificial or unlikely to occur in practice.

Given these insights, what can we say about the distinctions that *are* sufficient for calculating the optimal policy? Distinguishing between states that have different policy actions is necessary—but the proof shows us that it is not sufficient. The agent must also distinguish states with different utilities.

4.2.2 The Utile Distinction Test

The utile distinction test is defined as follows:

> The utile distinction test distinguishes states that have different policy actions or different utilities, and merges states that have the same policy action and same utility.

(When the agent is not given a model of the environment, the agent will have to estimate a model using data it gathers from experience. A statistical test will be needed to determine if the data is sufficient to claim statistical significance in utility differences.)

A proof that the utile distinctions are the set of distinctions that are necessary and sufficient for learning the optimal policy would be very interesting. However, I currently have no such proof.

4.3 Utile Distinction Memory Algorithm

The utile distinction test could be applied to many different types of state representations. The Utile Distinction Memory algorithm applies the test to a partially observable Markov decision process (POMDP).

The agent uses the Baum-Welch procedure [Rabiner, 1989] to learn the parameters of the POMDP. Baum-Welch is an Expectation-Maximization (EM) algorithm—an algorithm in which the learner alternates between gathering statistics using the current model, and changing the current model based on the statistics [Dempster *et al.*, 1977]. The agent begins with a random or minimal model, then alternately: (1) performs trials consisting of N steps in order to gather statistics on the current model (the estimate step), and (2) uses those statistics to update the model (the maximization step). A large number of steps in each trial is required to make the experience statistically significant. The standard Baum-Welch procedure is defined on hidden Markov models—which can be seen as POMDP's without actions. Straightforward extensions allow the application of Baum-Welch to POMDP's.

At the heart of the Utile Distinction Memory algorithm is the utile distinction test. After the maximization step, that agent uses the test to determine whether it should split or join any model states. Splitting a state creates a new agent-internal state distinction; splitting can give new short-term memory to the agent when the transitions leading into the old state are divided differently among the new states. Joining states removes an agent-internal state distinction; two states that were previously distinguished will not be after being joined. In the standard implementation of Baum-Welch, the number of model states is fixed before learning begins; in UDM, the agent dynamically changes the number of model states according to how many states the utile distinction test says are beneficial for predicting reward.

The utile distinction test is implemented as follows: UDM keeps future discounted reward information associated with the incoming transitions to a state. If the state satisfies the Markov property then all the reward values should be similar; however, if there are statistically significant differences in the rewards, then the identity of the state the agent comes from *is* significant to predicting reward, the state must be non-Markovian, and splitting this state would help the agent more consistently predict reward.

One split may allow further nested distinctions because it will create new separate transitions into other states. In this way, it is possible for UDM to build multi-step memories. It is, however, difficult for UDM to discover the utility of multi-step memories. Section 4.6 discusses this problem in detail, and points towards the future chapters that solve this problem.

4.4 Details of the Algorithm

A partially observable Markov decision process is comprised of a finite set of states, $\mathcal{S} = \{s_1, s_2, ..., s_N\}$, a finite number of observations (percepts), $\mathcal{O} = \{o_1, o_2, ..., o_M\}$ and a finite set of actions $\mathcal{A} = \{a_1, a_2, ..., a_K\}$. For each state there is a vector of observation probabilities—we write $O(o_i|s_j)$ for the probability of seeing observation o_i while in state s_j. For each state-action pair there is a vector of transition probabilities—the notation $T(s_k|s_i, a_j)$ signifies the probability that executing action a_j from state s_i will result in state s_k. The agent's belief about the state of the world is represented by a vector of its state occupation probabilities, written $\vec{\pi}(t) = \langle \pi_1(t), \pi_2(t), ..., \pi_N(t)\rangle$, where $\pi_i(t)$ is the agent's belief that the world state is represented by s_i at time t.

To calculate the state occupation probabilities at time $t+1$, the agent uses the "forward" part of the Baum-Welch forward-backward procedure [Rabiner, 1989], using actions it has generated with its current policy

$$\pi_j(t+1) = k \cdot O(o_{t+1}|s_j) \sum_i T(s_j|s_i, a_t)\, \pi_i(t), \tag{4.2}$$

where k is whatever constant is needed to make $\sum_i \pi_i(t+1) = 1$, and o_{t+1} is the agent's observation at time $t+1$, and a_t is the action executed by the agent at time t.

While choosing actions and altering state-action values with the reward from actions, UDM uses Q-learning superimposed on hidden Markov models, as in [Chrisman, 1992a]. We write $q(s_i, a_j)$ for the state-action value of action a_j specific to model state s_i. The agent obtains Q-values representative of the agent's current belief about the world state from the state-action values kept in the HMM states by letting all the HMM states "vote" in proportion to their occupation probability:

$$Q(\vec{\pi}(t), a_j) = \sum_i \pi_i(t)\, q(s_i, a_j) \tag{4.3}$$

This is an approximation. Formally, the agent should use belief states as the domain for the Q-functions [Littman, 1994b]. By using this approximation we loose the ability to represent more than one-step uncertainty about the agent's current state. This approximation, however, greatly simplifies the problem because belief state space is continuous and high-dimensional.

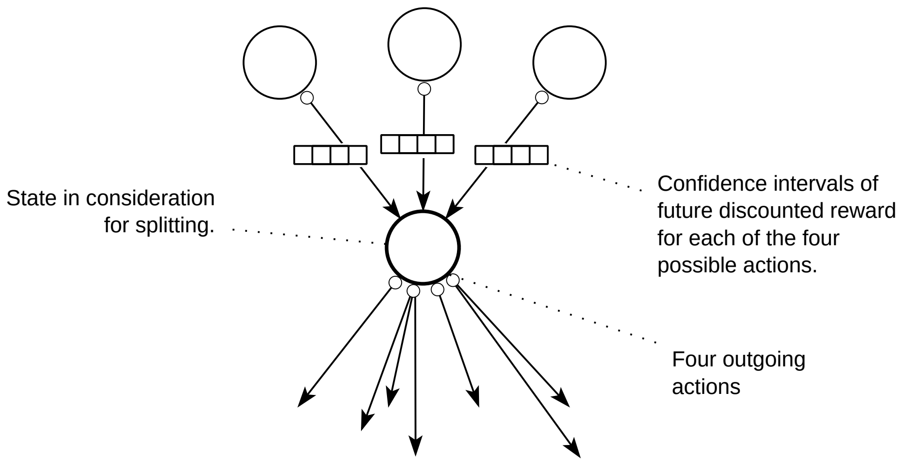
State in consideration for splitting.
Confidence intervals of future discounted reward for each of the four possible actions.
Four outgoing actions

Figure 4.2: A state in consideration for splitting. Associated with incoming transitions UDM keeps confidence intervals of future discounted reward for each action executed from the state in question.

After calculating Q-values, the agent then chooses the action with the largest Q-value, ($i.e.$, $f(\vec{\pi}) = \text{argmax}_a Q(\vec{\pi}, a)$. The state-action values are updated according to the standard Q-learning rule, again modified to use the state occupation probabilities:

$$\begin{aligned} \forall s_i,\ q(s_i, a_t) &= (1 - \beta\pi_i(t))q(s_i, a_t) + \\ &\quad \beta\pi_i(t)(r_t + \gamma U(\vec{\pi}(t+1)) \qquad (4.4) \\ U(\vec{\pi}(t+1)) &= \max_a Q(\vec{\pi}(t+1), a) \qquad (4.5) \end{aligned}$$

where $\beta, 0 < \beta < 1$, is the learning rate, $\gamma, 0 < \gamma < 1$, is the temporal discount factor, and $U(\vec{\pi}(t+1))$ is the expected utility of the next state given the agent's beliefs about which states it may be in. Effectively the state occupation probability, $\pi_i(t)$, becomes part of the learning rate. Thus far we have not diverged from [Rabiner, 1989] and [Chrisman, 1992a]. Next I describe the part of the algorithm specific to UDM.

With the incoming transitions to a state, UDM keeps statistics on the future discounted reward received after leaving the state in question. Future discounted reward is also called *return*. If the state is Markovian with respect to return, then the return values on all the incoming transitions should be similar. However if there are statistically significant differences between any of the incoming transitions, splitting the state and appropriately dividing the transitions between the split states will help the agent predict return.

Since two perceptually aliased states in the world may have the same utility, but require different actions, UDM actually keeps separate statistics for the different actions executed from the HMM state.

Statistical significance is tested using confidence intervals, which we calculate using the same method described in Kaelbling's dissertation [Kaelbling, 1990b]. For each interval we keep a running count, n, a sum of values, $\sum x$, and a sum of the squares of the values, $\sum x^2$. The upper and lower bounds are then calculated by

$$\bar{x} \pm t_{\alpha/2}^{(n-1)} \frac{s}{\sqrt{n}} \tag{4.6}$$

where $\bar{x} = (\sum x)/n$ is the sample mean, and

$$s = \sqrt{\frac{n \sum x^2 - (\sum x)^2}{n(n-1)}} \tag{4.7}$$

is the sample standard deviation, and $t_{\alpha/2}^{(n-1)}$ is the Student's t function with $n-1$ degrees of freedom. The parameter α, determines the confidence with which values will fall inside the interval.

These return statistics kept with transitions are never used to change the agent's state-action values, $q(i, A)$; they are only used to determine when and how to split a state.

The agent begins with a fully connected hidden Markov model containing a state for each percept. The observation probabilities, $O(o_i|s_j)$, are preset such that each percept is biased toward a different state.[1] The transition probabilities, $T(s_k|s_i, a_j)$, are all made equal. The agent then executes a series of m-step trials, (where m must be chosen so that is is large enough to gather enough experience for the statistical tests to follow). During each trial it updates the $q(s_i, a_j)$ values according the equations above, and also keeps a record of the actions, percepts, and rewards in history arrays $\mathtt{A}[t]$, $\mathtt{P}[t]$ and $\mathtt{r}[t]$.

At the end of each m-length trial the agent runs the tests for determining what states, if any, should be split. This begins by performing the Baum-Welch procedure for updating the model parameters. The procedure improves the model's ability to use percepts and actions to distinguish the agent's current state, but since the test used for splitting states is separate from Baum-Welch, the agent will not create new memory capacity in order to predict perception. In the process of the Baum-Welch procedure the agent calculates improved estimates of the state occupation probabilities over time, $\pi_i(t)$, and estimates of the transition occupation probabilities over time, $\xi(s_i, a_j, s_k|t)$, (the probability that at time t the agent passed from state s_i to state s_k using action a_j). See [Rabiner, 1989] for details. Then the agent calculates return values over time, $\mathtt{return}[t]$, using the reward history, $\mathtt{r}[t]$:

```
return$[m]$ = r$[m]$
for $t = m - 1$ to 0
    return$[t]$ = r$[t] + \gamma \cdot$ return$[t+1]$
```
(4.8)

where γ is the temporal discount factor. Because any state could be aliased and could have state-action values resulting from a meaningless combination rewards from different

[1] UDM has the ability to discover beneficial memory distinctions, but not perceptual distinctions, thus the model is initialized to distinguish between all percepts.

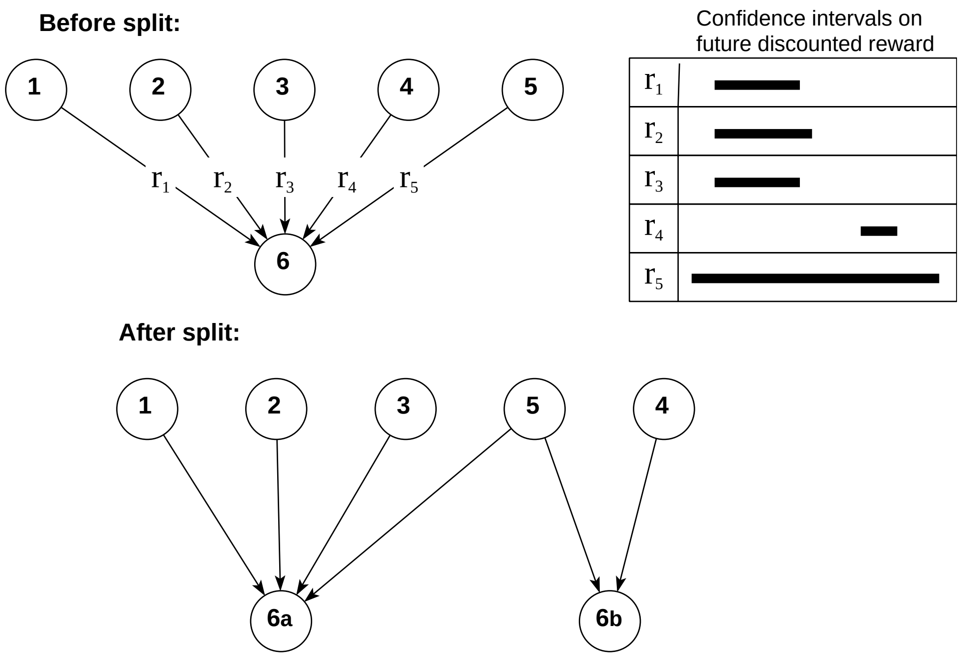

Figure 4.3: Determining when and how a state should be split by using confidence intervals on future discounted reward. Consider all incoming transitions to state 6, and compare the set of confidence intervals corresponding to each action executed from state 6. If any of the intervals do not overlap, then the agent has determined that by treating state 6 differently depending on which state it came from it can get a statistically significant increase in its ability to predict reward. The set of intervals is divided into clusters of overlapping intervals, and each cluster is assigned to a different copy of state 6. The confidence interval table in the upper right only shows the intervals for one outgoing action.

world states, the state-action values (and state utilities defined as the maximum over the state-action values) cannot be trusted, and only reward directly from the world is used to calculate return.

Now the agent has all the information necessary to assign return values to specific transitions. The agent considers the value of `return`$[t]$ at each time step and includes this value in the statistics for each transition in proportion to the transition occupation probability at the previous time step:

```
for $t = 1$ to $m - 1$
    for all transitions, trans, that use action $\mathtt{A}[t-1]$
        lr = $\beta \cdot \xi_{\mathtt{trans}}[t-1]$
        trans.count$_{\mathtt{A}[t]}$ += lr
        trans.sum$_{\mathtt{A}[t]}$ += lr $\cdot$ return$[t]$
        trans.sumsquares$_{\mathtt{A}[t]}$ += lr $\cdot$ (return$[t]$)$^2$
```
(4.9)

where β is the agent's learning rate, `lr` is the learning rate for a particular transition at a particular time, and $\xi_{\texttt{trans}}[t-1]$ is the transition occupation probability $\xi(s_i, a_j, s_k|t-1)$ at time $t-1$ of the particular transition, `trans`, belonging to source state s_i, action a_j and destination state s_k. The `trans`-dot quantities in the last three lines refer to the three statistics necessary for computing the upper and lower bounds; `trans.count` is the count, n, `trans.sum` is the sum, $\sum x$, and `trans.sumsquares` is the sum of squares, $\sum(x^2)$. The `A` subscripts indicate that they are indexed by the different actions executed in the destination state of the transition.

Using the count, sum, and sum of squares the agent can obtain upper and lower bounds of the return value confidence intervals for each transition. The agent determines if the differences in return on two incoming transitions are statistically significant by measuring whether or not their confidence intervals overlap. When any two of the confidence intervals with the same outgoing action fail to overlap the state is split.

The agent divides the set of incoming transitions into disjoint subsets, such that all transitions in a subset overlap with each other and each subset is as large as possible. To perform this clustering optimally is actually an NP-complete problem (Minimum Cover, [Garey and Johnson, 1979]), but in practice, greedy solutions are adequate. After the clustering, the state is duplicated (including incoming and outgoing transitions) enough times so that there is one copy per cluster. Then, any incoming transitions that are not elements of a duplicate state's assigned cluster are removed.

States may also be joined if the union of their incoming transitions all overlap with each other, and if one set of incoming transitions is a subset of the other. If the analysis at the end of any m-length trial results in any splits or joins, the `trans`-dot statistics are re-initialized.

4.5 Experimental Results

UDM has been demonstrated working in a maze task where perception is defined by immediately adjacent barriers. The Local Perception Grid World (LPGW) is based on simulated worlds used by Sutton, Whitehead, Thrun and others [Sutton, 1990b; Whitehead, 1992; Thrun, 1992a]. The agent moves about a discrete grid by executing the actions for moving north, east, south, and west. Whenever the agent attempts to move into a grid occupied by a barrier, the agent remains where it is and receives a reward of -1.0. Whenever the agent moves into any of the one or more "goal" squares, it receives a reward of 1.0. For all other actions the agent receives a reward of -0.1. After reaching the goal square the agent is transported to another randomly chosen location in the maze.

The difference between the LPGW and the other grid worlds is that instead of defining perception as the unique row and column numbers of the agent's position (a perception that does not cause perceptual aliasing), LPGW defines perception to be a bit vector of length four in which the bits specify whether or not there is a barrier to the agent's immediate north, east, south and west. This perception is rich with perceptual aliasing possibilities (and is also a little more realistic for a navigating robot one might build).

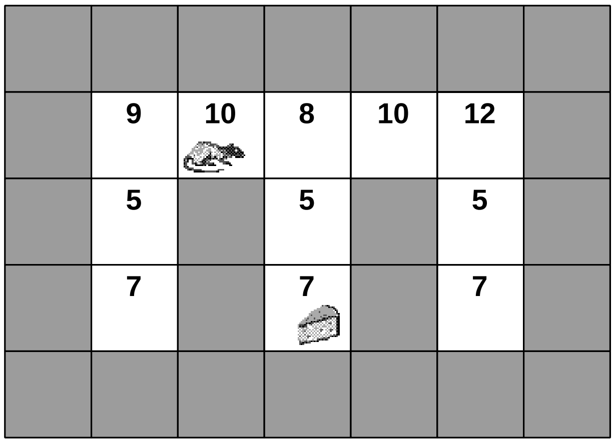

Figure 4.4: One of several mazes successfully solved by UDM. States labeled by the same integer are perceptual aliased. This particular maze shape is called the M-maze.

Several experiments also included noise in the environment. With probability 0.1 the agent perceives some other perception vector than its current world position would have specified. Also with probability 0.1 the agent's chosen actions are randomly changed to one of the four.

One maze solved by UDM appears in Figure 4.4. The numbers in the squares are the decimal equivalents of the bit vectors interpreted as binary numbers. Although the three state 5's are all perceptually equivalent, the agent must learn to go south in the 5 in the center, and go north in the 5's on the right and left. The hidden Markov model built by UDM has two states with high probabilities of observing 5—one representing the 5 in the center and the other combining the two 5's on the right and left. Percept 10 is also aliased, and is split in two since different actions are required in the two world states.

Without noise, UDM consistently learned the optimal policy for this maze in five trials of 500 steps each. The agent used a learning rate of $\beta = 0.6$, a temporal discount factor of $\gamma = 0.7$, a confidence on the upper and lower bounds of $(1 - \alpha) = 0.95$, and a random action probability (exploration rate) of 0.1. The sequence of HMM's created appear in Figure 4.5. Notice that percept 5 is represented by only two states even though there are three world states that produce percept 5. On the sides, where the same action is required, UDM keeps the two world state 5's aliased to the same internal state. With noise, UDM learned the same optimal policy, but the agent required 15 trials of 500 steps each.

Figure 4.5 shows diagrams of the POMDP built to solve this maze. The HMM states are fully connected, but the diagrams only show transitions with probability greater than 0.3. Also not shown here are the different actions that cause the transitions.

Trial 1: The agent has learned some transition probabilities, but it has not yet gathered enough statistics to split any states.

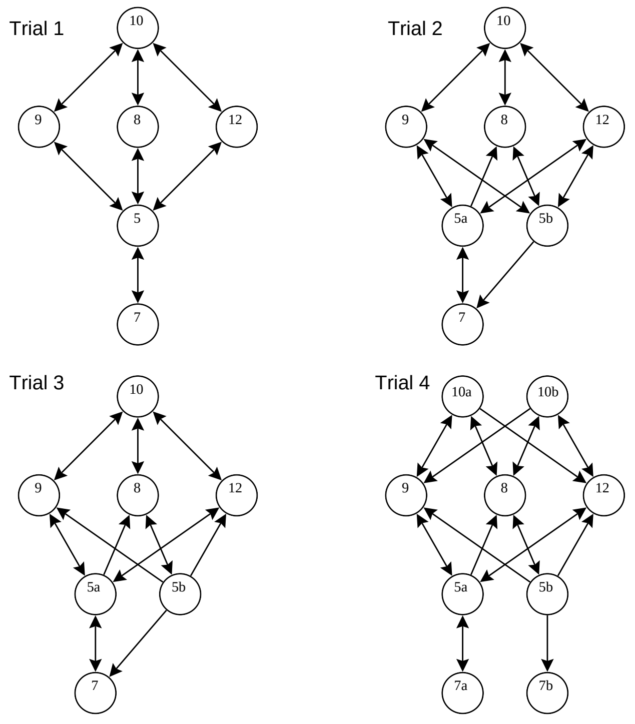

Figure 4.5: The sequence of hidden Markov models created by UDM as it learns the task shown in Figure 4.4.

Trial 2: The agent has discovered that the future discounted reward received after leaving state 5 is significantly different when it arrives from state 8 than it is when it arrives from state 7. UDM has duplicated state 5, creating 5a and 5b, then removed the state 8 incoming transition from 5a and removed the state 7 incoming transition from 5b.

Trial 3: The agent has found non-overlapping confidence intervals on the transitions into 5b—arriving from state 8 is significantly different than arriving from states 9 or 12. UDM temporarily splits 5b into 5b and 5c, but then 5c is joined with 5a.

Trial 4: Two splits occur at the end of this trial. Average future reward after leaving the center state 7 is greater than after leaving the side state 7's because from the center state 7 the agent is teleported to locations that are are often closer to the goal. UDM splits state 7 recognizing that arriving from 5a is different than arriving from 5b. State 10 is also split because the confidence intervals for leaving 10 by going east and leaving 10 by going west don't overlap in the transitions coming from states 9 and 12. The agent will now arrive in state 10a from state 9 and arrive in state 10b from state 12. There are no more changes to the HMM after trial 5.

4.6 Discussion

The Utile Distinction Memory technique provides a refutation of the "Utile-Distinction Conjecture" [Chrisman, 1992a], which stated that is was impossible to introduce only memory distinctions that impact utility. As such, UDM acts as a starting point from which to design improved learners that use memory to create selective, task-specific representations of their environment.

UDM is not without limitations and problems. The computational requirements for the state splitting test are significant, in terms of both storage and computation. Any method that uses the Baum-Welch procedure requires $O(KN^2m)$ time and storage. As long as $K < m$, UDM does not increase these requirements. However , the increased constants are significant.

Even more harmful than the requirements for significant computation, are UDM's requirements for a large amount of experience. The Baum-Welch procedure requires a large amount of experience for each trial because it must gather reliable statistics for adjusting the model. Only one or a few state splits are made after each trial because the need for some splits may not be apparent until other splits are made. The combination of these factors result in an extremely large number of training steps before the task is learned.

UDM splits states in order to increase memory-based distinctions, but this implementation has no method for splitting states to increase perceptual distinctions that predict future reward. For this reason UDM begins with one state per percept, a strategy that will obviously not work for large perception spaces. Some method for making perceptual distinctions will be necessary. For example, Chapman and Kaelbling's G algorithm [Chapman and Kaelbling, 1991] or some technique based on statistics for the unused perception bits in a state can work in conjunction with UDM. Chapter 7 describes an algorithm that makes both history and perceptual distinctions.

Although UDM can build memory chains of arbitrary length, it does require that some statistically significant benefit be detectable for each split individually in sequence. UDM has solved mazes with multi-step paths between the reward-predicting percepts and the reward, however, it would not be able to solve problems in which a conjunction of percepts in sequence predicted reward, but each of the percepts on its own was independent of future reward. This assumption about the detectable relevance of pieces of information in isolation is also made by [Chapman and Kaelbling, 1991] and [Maes and Brooks, 1990]; additionally, it is a key factor in Genetic Algorithms [Holland, 1975; Booker *et al.*, 1989].

5 Instance-Based Learning

Chapter Outline

This chapter discusses the similarity between learning with hidden state and learning in continuous state spaces, and explains the way in which instance-based learning is particularly appropriate to both. It presents Nearest Sequence Memory, an algorithm based on k-nearest neighbor. The algorithm can easily build multi-step memories, and learns roughly an order of magnitude faster than several other algorithms, and thus overcomes the chief drawback of Utile Distinction Memory.

The previous chapter presented Utile Distinction Memory, an algorithm that uses state splitting tests on partially observable Markov decision processes in order to make utile distinctions in short-term memory. The algorithm, however, has two key drawbacks: it learns extremely slowly, and it has trouble discovering the utility of memories longer than one time step since the statistical test only examines the benefit of making a single additional state split at a time.

This chapter advocates a new method for reinforcement learning with short-term memory that addresses both these concerns. It introduces a family of methods that I term *instance-based state identification.* The approach was inspired by the successes of instance-based (also called "memory-based") methods for learning in continuous perception spaces, (*i.e.* [Atkeson, 1992; Moore, 1992; Schneider, 1995; Moore *et al.*, 1995]). The key feature of instance-based learning methods is that they explicitly keep all the raw data they are shown.

The first implementation of Instance-Based State Identification finds multi-step memories easily and learns with about an order of magnitude fewer steps than several previous reinforcement learning algorithms.

5.1 The Key Idea: Instance-Based Learning for Variable Granularity

The application of instance-based learning to short-term memory for state identification is driven by the important insight that learning in continuous spaces and learning with hidden state share a crucial feature: they both begin learning without knowing the final granularity of the agent's state space. The former learns which regions of continuous input space can be represented uniformly and which areas must be finely divided among many states. The later learns which perceptions can be represented uniformly because they uniquely identify a course of action without the need for memory, and which perceptions must be divided among many states (each with their own detailed history to distinguish them from other perceptually aliased world states). The first approach works with a continuous geometrical input space, the second works with a action-percept-reward "sequence" space, (or "history" space). Large continuous regions correspond to less-specified, small memories; small continuous regions correspond to more-specified, large memories.[1]

Furthermore, learning in continuous spaces and sequence spaces both have a lot to gain from instance-based methods. In situations where the state space granularity is unknown, it is especially useful to memorize previous raw experiences. If the agent incorporates experience merely by averaging it into its current, flawed state space granularity, it is bound to attribute experience to the wrong states; experience attributed to the wrong state turns to garbage and is wasted. When faced with an evolving state space, keeping previous raw experience is the path of least commitment, and thus the most cautious about losing information.

Keeping records of previous raw experience does incur certain expenses in terms of computation time and storage, but these expenses can often be justified. For instance, compare the time required to perform a million multiplications with the time it takes the robot to move down the hallway. Often learning trials are more expensive than computation—expensive in many ways: wall clock time, power consumed, danger to the robot and danger to the surroundings, for example. Techniques such as DYNA [Sutton, 1990b], Experience Replay [Lin, 1991b], Transitional Proximity Q-learning [McCallum, 1992] and Asynchronous Dynamic Programming [Barto *et al.*, 1991] are all examples of efforts to substitute world experience with storage and computation.

[1]This analogy between continuous spaces and memory spaces provides a further perspective against implementing memory with constant-sized perception windows: Constant-sized perception windows are undesirable for precisely the same reason constant-sized discretizations of continuous spaces are undesirable. If the discretization is too coarse-grained, the agent will fail; if the discretization is too fine-grained the number of states is exponentially more than needed; different granularity is needed in different regions of state space.

5.2 Nearest Sequence Memory Algorithm

There are many possible instance-based techniques to choose from, but I wanted to keep the first application simple. With that in mind, this initial algorithm is based on k-nearest neighbor. I call it Nearest Sequence Memory, (NSM). It bears emphasizing that this algorithm is the most straightforward, simple combination of instance-based methods and history sequences that one could think of; there are still more sophisticated instance-based methods to try. The surprising result is that such a simple technique works as well as it does.

Figure 5.1 depicts the analogy between k-nearest neighbor in a continuous geometric space and k nearest neighbor in a sequence space. By using this neighborhood function, the agent finds situations from its past that are most similar to its current situation, where "most similar" not only includes the current percepts but also the percepts leading up to current situation. The agent thus has an internal state space with variable-length short-term memory because when choosing instances from which to extract expected reward values, it tends to prefer instances with better matching history.

Any application of k-nearest neighbor consists of three parts: 1) recording each experience, 2) using some distance metric to find neighbors of the current query point, and 3) extracting output values from those neighbors. We apply these three parts to action-percept-reward sequences and reinforcement learning by Q-learning [Watkins, 1989] as follows:

1. For each step the agent makes in the world, it records the action, perception and reward by adding a new state to a single, long chain of states. Thus, each state in the chain contains a snapshot of the agent's immediate experience at that moment, and all the agent's experiences are laid out in a time-connected history chain.

2. When the agent is about to choose an action, it finds states considered to be similar by looking in its state chain for states with histories similar to the current situation. The longer a state's string of previous experiences matches the agent's most recent experiences, the more likely the state represents where the agent is now.

3. Using the states, the agent calculates Q-values by averaging together the expected future reward values associated with the k nearest states for each action. The agent then chooses the action with the highest Q-value. The regular Q-learning update rule is used to update the k states that voted for the chosen action.

Choosing to represent short-term memory as a linear trace is a simple, well-established technique. Order-n Markov chains use this representation; the fixed-sized time windows used by Lin and Albus are examples of order-n Markov chains [Lin, 1993; Albus, 1991]. Nearest Sequence Memory also uses a linear trace to represent memory, but it differs from these fixed-sized window techniques because it provides a *variable* memory-length—like k-nearest neighbor, NSM can represent varying resolution in different regions of state space. When only short strings match, this corresponds to a

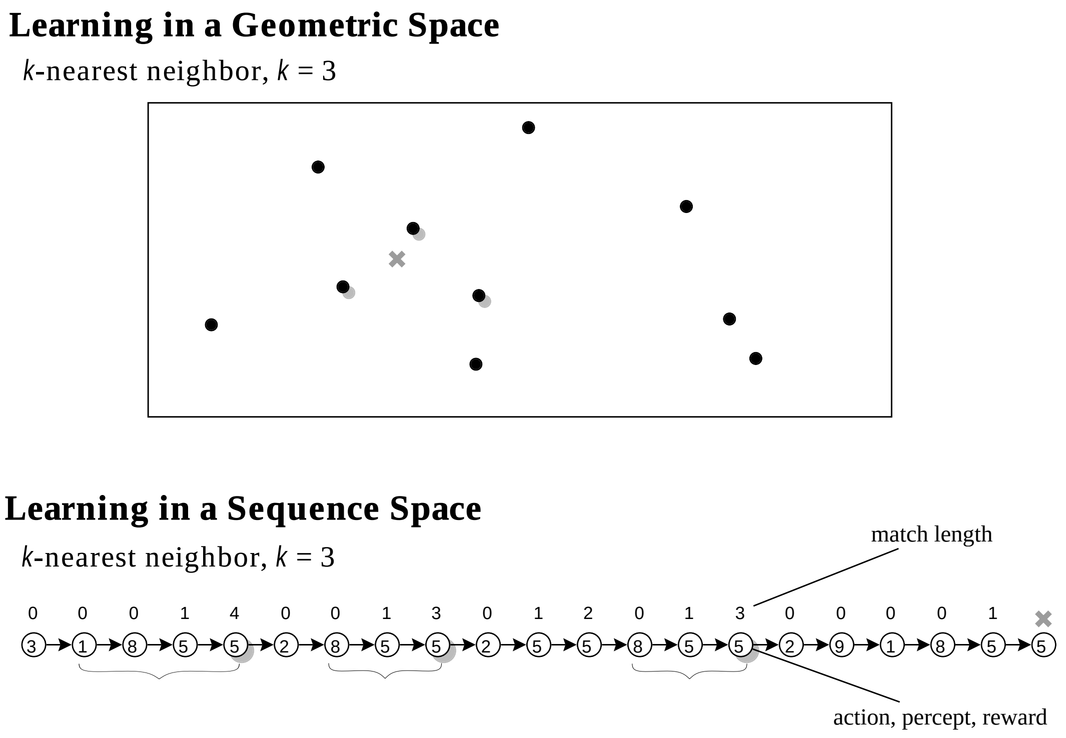

Figure 5.1: Comparing k-nearest neighbor in a continuous space and in a sequence space. Both have a database of instances, and both have a point which we want to query. In each case, the "query point" from which we measure neighborhood relations is indicated with a gray cross, and the three nearest neighbors are indicated with gray shadows. In a geometric space (shown on top), instances are the black dots, each indicating some vector of values in multi-dimensional space. The neighborhood metric is defined by Euclidean distance. In a sequence space (shown on bottom), instances are circles connected by arrows, each indicating an action/percept/reward snapshot in time, with all the snapshots laid out in time sequence order. (The numbers in the circles indicate action/percept/reward triples.) The neighborhood metric is determined by sequence match length, that is, the number of preceding states that match the states preceding the query point.

large, unspecific partition of state space; when long strings match, this corresponds to a small partition of state space (a specific short-term memory).

5.3 Details of the Algorithm

The interaction between the agent and its environment is described by actions, percepts and rewards. There is a finite set of possible actions, $\mathcal{A} = a_1, a_2, ..., a_m$, a finite set of possible percepts (observations), $\mathcal{O} = o_1, o_2, ..., o_n$, and scalar range of possible rewards, $\mathcal{R} = [x, y], x, y \in \Re$. At each time step, t, the agent executes an action, $a_t \in \mathcal{A}$, then as a result receives a new percept, $o_t \in \mathcal{O}$, and a reward, $r_t \in \mathcal{R}$.

We say that the percept "results" from the action in order to be consistent with models of *active* perception in which the percept is a function of both the world state and the previous action [Ballard and Brown, 1992; McCallum, 1993a].

Note also the atypical grouping of the action, percept and reward in time. In many formulations, the action is given the same time index as the percept and reward that preceded it, *i.e.*, $(o_{t-1}, r_{t-1}, a_{t-1}, o_t, r_t, a_t, o_{t+1}, r_{t+1}, a_{t+1}, ...)$. The explanation of NSM is made simpler by giving the action the same time index as the percept and reward that *follow* it, *i.e.*, $(o_{t-1}, r_{t-1}, a_t, o_t, r_t, a_{t+1}, o_{t+1}, r_{t+1}, a_{t+2}, ...)$. In this chapter we use the later grouping.

Subscripts on the symbols a, o, and s indicate the time step associated with specific actions, percepts and states. The subscript t refers the the current time step; the subscripts i and j refer to arbitrary time steps.

Just like other instance-based algorithms, Nearest Sequence Memory records each of its raw experiences. The action and environmental state at time t is captured in a "state" data point, written s_t. Recorded with that state is all the available information associated with time t: the action, a_t, the resulting percept, o_t and resulting reward, r_t. Also associated with state s_t is a slot to hold a single expected future discounted reward estimate, denoted $q(s_t)$. This value is associated with action a_t and no other action. We use the notation q to indicate the expected reward value associated with an individual instance; the notation Q refers to the average of the q-values extracted from the k neighboring instances that are closest to the query point.

The Nearest Sequence Memory algorithm consists of the following steps:

1. Find the k nearest neighbor (most similar) states for each possible future action. The state currently at the end of the chain is the "query point" from which we measure all the distances. The neighborhood metric is defined by "string match" length; it is the number of preceding experience records that match the experience records preceding the "query point" state. This metric can be defined recursively. (Here higher values of $n(s_i, s_j)$ indicate that s_i and s_j are closer neighbors; the notation a_{i-1} indicates the action associated with the state preceding state s_i in the chain.)

$$n(s_i, s_j) = \begin{cases} 1 + n(s_{i-1}, s_{j-1}), & \text{if } (a_{i-1} = a_{j-1}) \wedge (o_{i-1} = o_{j-1}) \wedge (r_{i-1} = r_{j-1}) \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

Considering each of the possible future actions in turn, we find the k nearest neighbors and give them a vote, $v(s_i)$. Ties are resolved by preferring states towards the end of the chain.

$$v(s_i) = \begin{cases} 1, & \text{if } n(s_t, s_i) \text{ is among the } k \max_{\forall s_j | a_j = a_i} n(s_t, s_j)\text{'s} \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

In the equation above, s_t is the current state, and the "if" expression indicates that a state gets vote 1 only if the neighborhood metric between it and the current state is among the k maximum when compared to the neighborhood similarity values of all the other states that have the same outgoing action.

2. Determine the Q-value for each action by averaging the q-values from the k voting states (states, s, with $v(s) = 1$) from which that action was executed.

$$Q_t(a_i) = \sum_{\forall s_j | a_j = a_i} (v(s_j)/k)\, q(s_j) \tag{5.3}$$

3. Select an action by maximum Q-value, or by random exploration. According to an exploration probability, e, either let a_{t+1} be randomly chosen from $\mathcal{A}$, or

$$a_{t+1} = \text{argmax}_a Q_t(a) \tag{5.4}$$

4. Execute the action chosen in step 3, and record the resulting experience. Do this by creating a new "state" representing the current state of the environment, and storing the action-percept-reward triple associated with it:

   Increment the time counter: $t \leftarrow t + 1$. Create s_t; record in it a_t, o_t, r_t.

   The agent can limit its storage and computational load by limiting the number of instances it maintains. Instead of allowing the number of instances to grow indefinitely with the number of experience steps, the agent can choose to keep not more than N of them, where N is some reasonably large number like 1000. Once it reaches N instances, it would, with each step, discard the oldest instance and add the new one. The choice of N is balanced by the desire to limit storage and computation, on the one hand, versus the need for the agent to remember experiences from different parts of the environment it visits, on the other. This also provides a way to handle a changing environment.

5. Update the q-values by vote. Perform the dynamic programming step using the standard Q-learning rule to update those states that voted for the chosen action. Note that this actually involves performing steps 1 and 2 to get the next Q-values

needed for calculating the utility of the agent's current state, U_t. (Here β is the learning rate.)

$$\begin{aligned} U_t &= \max_a Q_t(a) && (5.5) \\ (\forall s_i | a_i = a_{t-1}) \;\; q(s_i) &\leftarrow (1 - \beta v(s_i)) q(s_i) + \beta v(s_i)(r_i + \gamma U_t) && (5.6) \end{aligned}$$

Note the use of r_i instead of r_t in the update of $q(s_i)$. This helps the agent estimate an accurate average reward in a stationary world—using r_t would have biased the average toward recent rewards, where the strength of the bias would have been dictated by the learning rate, β.

The agent augments the above one-step Q-learning by also passing discounted reward backwards through its recorded history chain for as long as the propagation increases the states' Q-values. In practice, this does not occur often. The process could be described as a kind of "conservative" TD-λ where $\lambda = 1$.

Nearest Sequence Memory performs two kinds of learning. Like all instance-based methods, it learns by populating its input space with raw experience. Unlike most other instance-based methods, it must also perform dynamic programming on values from raw experience. Usually instance-based methods are used as function approximators whose required outputs are the values provided by raw experience—they are supervised learners. In Nearest Sequence Memory, the required output values are Q-values, which are not provided by raw experience, but related to the raw rewards through dynamic programming—Nearest Sequence Memory is a reinforcement learner. These two kinds of learning need not always occur together. For instance, the agent could stop adding states to the chain, but continue to execute steps in the world and do the dynamic programming as implemented by Q-learning. Section 7.2.3 discusses this distinction further.

5.4 Experimental Results

This section shows experimental results comparing the performance of Nearest Sequence Memory to three other algorithms. Performance is demonstrated using the tasks chosen by the other algorithms' designers. In each case, NSM learns the task with roughly an order of magnitude fewer steps. Although NSM learns good policies quickly, it does not always learn optimal policies. Section 5.5.2 discusses why the policies are not always optimal and how NSM could be improved.

5.4.1 Perceptual Distinctions Approach

The Perceptual Distinctions Approach [Chrisman, 1992a] was demonstrated in a space-ship docking application with hidden state. The task was made difficult by noisy sensors and unreliable actions. Some of the sensors returned incorrect values 30 percent of the time. Various actions failed 70, 30 or 20 percent of the time, and when they failed, resulted in random states.

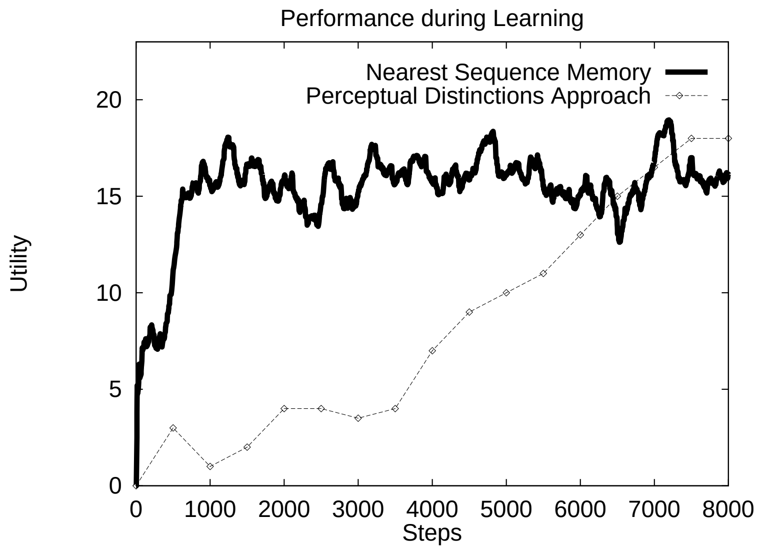

Figure 5.2: Comparing Perceptual Distinctions Approach and Nearest Sequence Memory on Chrisman's Docking Task. The value plotted is the utility (sum of future discounted rewards) of the states the agent's policy visits.

The Perceptual Distinctions Approach used learning rate $\beta = 0.1$, temporal discount factor $\gamma = 0.9$, an Expectation-Maximization (EM) cycle of 1000 steps, and an exploration probability $e = 0.1$. Nearest Sequence Memory used learning rate $\beta = 0.1$, temporal discount factor $\gamma = 0.9$, maximum instance chain length $N = 1000$, exploration probability $e = 0.1$, and number of neighbors, $k = 8$. A higher k was chosen for this experiment than for the others because of the noisy environment. A graph of the performance during learning appears in figure 5.2. Performance is indicated using the measure that the agent itself is trying to maximize: sum of future discounted reward (utility). The NSM results have been averaged over five runs. The Perceptual Distinctions Approach takes almost 8000 steps to learn the task. Nearest Sequence Memory learns a good policy in less than 1000 steps, although the policy is not quite optimal.

Not only does NSM learn the task in fewer steps than the Perceptual Distinctions Approach, it also learns the task with less computation. The Baum-Welch procedure [Rabiner, 1989] is an integral part of the Perceptual Distinctions Approach, and Baum-Welch takes time and space $O(|\mathcal{S}|^2|\mathcal{A}|N)$, where $|\mathcal{S}|$ is the number of states in the model, $|\mathcal{A}|$ is the number of actions and N is the number of steps in the EM cycle. NSM takes time $O(N)$, where N is the maximum instance chain length. For both algorithms N is 1000.

5.4.2 Utile Distinction Memory

Utile Distinction Memory [McCallum, 1993b] was demonstrated on several local perception mazes. Unlike most reinforcement learning maze domains, the agent does not perceive global row and column numbers, but only indications of whether there is a barrier to the immediately adjacent up, down, right or left. Therefore, many maze positions are perceived as identical—including positions from which the agent must execute different actions. Each time the agent reaches the goal, it is reset to a random location in the maze.

Utile Distinction Memory used a learning rate $\beta = 0.9$, temporal discount factor $\gamma = 0.9$, and an Expectation-Maximization cycle of 500 steps. Nearest Sequence Memory used learning rate $\beta = 0.9$[2], temporal discount factor $\gamma = 0.9$, maximum chain length $N = 1000$, exploration probability $e = 0.1$, and number of neighbors $k = 4$. A graph of the number of steps required for learning appears in figure 5.3. The results are averaged over ten runs. In two of the mazes, Nearest Sequence Memory learns the task in only about 1/20th the time required by Utile Distinction Memory; in the other two, Nearest Sequence Memory learns mazes that Utile Distinction Memory did not solve at all. Although UDM has in principle the capability to learn multi-step memories, in practice they are difficult for UDM to discover [McCallum, 1993a]. NSM solves these multi-step memory tasks without problem.

Like the Perceptual Distinctions Approach, UDM uses the Baum-Welch procedure. Thus, UDM also has worse computational complexity than NSM.

5.4.3 Recurrent-Q

Recurrent-Q [Lin, 1993] was demonstrated on a robot two-cup retrieval task. The environment is deterministic, but the task is made difficult by two nested levels of hidden state and by providing no reward until the task is completely finished.

Lin actually tried three different neural network algorithms on this task: Window-Q, Recurrent-Q, and Recurrent-Model; Recurrent-Q is the one that learned this task in the fewest trials. Recurrent-Q used temporal discount $\gamma = 0.9$, and a trial time-out of 60 steps. It also used TD-λ ($\lambda = 0.8$) with Experience Replay, which is an experience recording and playback technique developed by Lin for speeding up learning. Nearest Sequence Memory used a trial time-out of 60 steps, learning rate $\beta = 0.9$, temporal discount factor $\gamma = 0.9$, maximum instance chain length $N = 1000$, exploration probability $e = 0.1$, and number of neighbors $k = 4$. A graph of performance during learning appears in figure 5.4. Nearest Sequence Memory learns good performance in about 15 trials, Recurrent-Q takes about 100 trials to reach equivalent performance.

A parallel processing implementation of Recurrent-Q could take computation time O(number of layers in the net). A parallel processing implementation of NSM could take computation time $O(1)$. Note also that, like NSM, Lin's Experience Reply mechanism requires the agent to store the chain of raw experience previous.

[2]In a deterministic environment the agent can handle a much higher learning rate than in the stochastic environment used above.

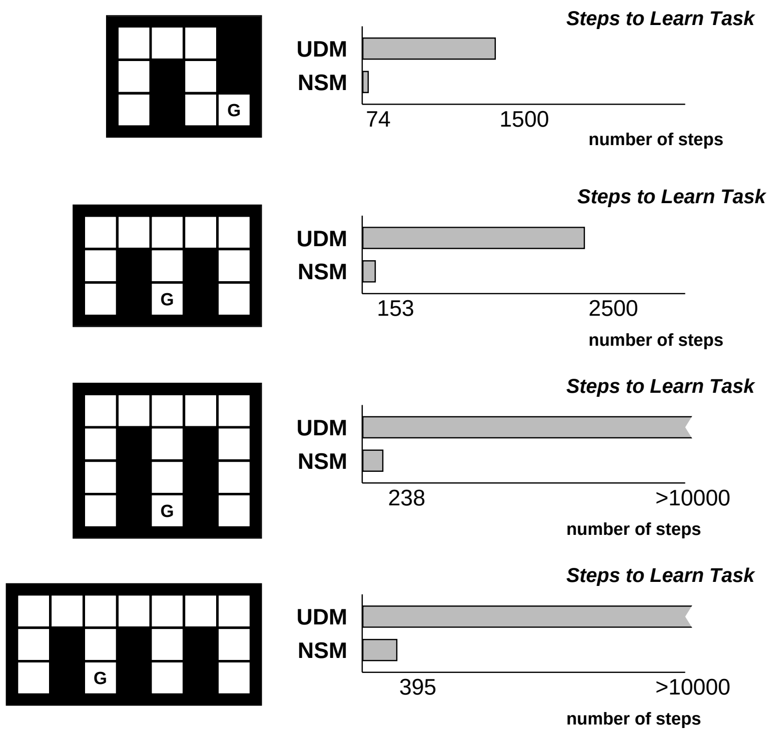

Figure 5.3: Comparing Utile Distinction Memory and Nearest Sequence Memory on a maze task. Many positions from which the agent should execute different actions are perceived as identical. The goal square is indicated with a “G”. The bar graphs show the number of training steps required to learn a perfect policy for all starting points. The results are averaged over ten runs.

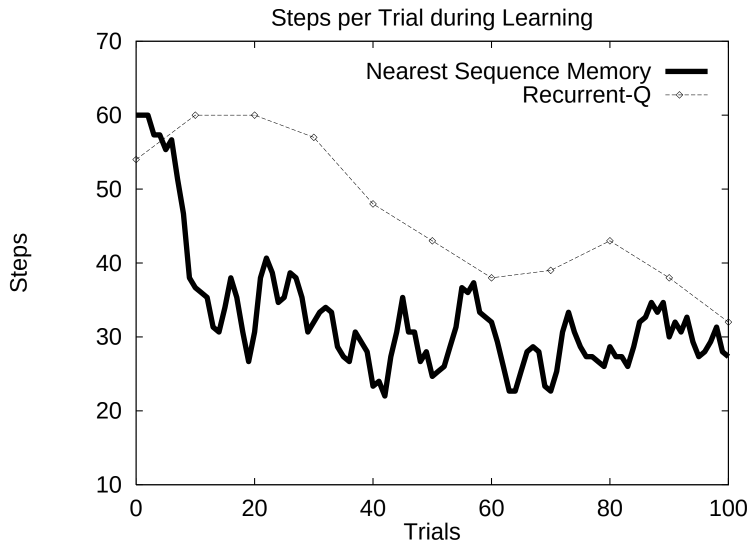

Figure 5.4: Performance comparison between Recurrent-Q and Nearest Sequence Memory on Lin's 2-cup task. (Lower is better.) The vertical axis shows the number of steps required to complete the 2-cup task twice, once from each of the two possible starting points. Each of these double task completions makes up a single trial. The horizontal axis measures number of trials since the beginning of learning. The learning curves show the mean results from five runs. The data from NSM is noisy because the lack of structure/noise distinction makes NSM's performance somewhat erratic; see section 5.5.2, "Structure/Noise Distinction."

5.5 Discussion

5.5.1 Reason for Better Performance

Nearest Sequence Memory offers much improved on-line performance and fewer training steps than its predecessors. Why is the improvement so dramatic? I believe the chief reason lies with the inherent advantage of instance-based state identification, as described in section 5.1, and restated in the following paragraph:

The key idea behind Instance-Based State Identification is the recognition that recording raw experience is particularly advantageous when learning to partition a state space, as is the case when the agent is trying to determine how much history is significant for uncovering hidden state. If, instead of using an instance-based technique, the agent simply averages new experiences into its current, flawed state space model, the experiences will be applied to the wrong states, and cannot be reinterpreted when the agent modifies its state space boundaries. Furthermore, data is always interpreted *in the context of the flawed state space*, always biased in an inappropriate way—not simply recorded, kept uncommitted and open to easy reinterpretation in light of future data.

This scenario is especially clear in the finite state machine approaches: experiences are averaged into the current FSM; after each state-splitting session, the agent re-gathers experience from scratch; the statistics used for learning are derived from the current, flawed FSM; and the raw data used to obtain these statistics is thrown away—not kept for reinterpretation in light of the new state space topology. The same statements apply to the recurrent neural network, except in this case the agent's internal state space is defined by the weights on connections to the memory-providing hidden units.

5.5.2 Areas for Improvement

The experimental results in this paper bode well for instance-based state identification. Nearest Sequence Memory is simple—if such a simplistic implementation works as well as it does, more sophisticated approaches may work even better. The following subsections discuss some ideas for improvement.

Better Distance Metric

The agent should use a more sophisticated neighborhood distance metric than exact string match length. A new metric could account for distances between different percepts instead of considering only exact matches. A new metric could also handle continuous-valued inputs.

There is, however, prior experience showing that neither of these improvements is strictly necessary for work with physical robots. The sequence matching algorithm implemented for *Toto* [Mataric, 1990] used a small, finite set of percepts and was completely intolerant to noise. The robot wandered rooms and hallways while avoiding obstacles, learned a map of the environment, and could use the map to reliably travel to arbitrary goal locations. *Toto*'s intolerant sequence matching algorithm overcame its quite noisy sensors by using an intermediate perceptual mechanism that translated the continuous, high-dimensional, noisy sensor values into elements of the useful percept set. Using a little domain knowledge and some averaging over time, the robot's intermediate layer produced noise-less percepts suitable for its intolerant string matching algorithm. A robot using Nearest Sequence Memory could make use of precisely the same mechanism. Furthermore, since NSM is somewhat tolerant of noisy percepts and actions, NSM could even continue to work if the intermediate perceptual mechanism were unreliable.

Unlike finite state machines, Nearest Sequence Memory cannot directly represent cycles in the environment. The agent should use well-chosen or learned abstract action primitives that encapsulate these loops.

Although the algorithms in the next two chapters do not follow this approach, we could imagine some technique based on variable weighted distance metric that made utile distinctions [Stanfill and Waltz, 1986].

Structure/Noise Distinction

Nearest Sequence Memory demonstrably solves tasks that involve noisy sensation and action, but it could perhaps handle noise even better if it used some technique for explicitly separating noise from structure.

K-nearest neighbor does not explicitly discriminate between structure and noise [Wettschereck and Dietterich, 1994; Deng and Moore, 1995]. If the current query point has neighbors with wildly varying output values, there is no way to know if the variations are due to noise, (in which case they should all be averaged), or due to fine-grained structure of the underlying function (in which case only the few closest should be averaged). Because Nearest Sequence Memory is built on k-nearest neighbor, it suffers from the same inability to methodically separate history differences that are significant for predicting reward and history differences that are not. I believe this is the single most important reason that Nearest Sequence Memory sometimes did not find optimal policies.

An interesting manifestation of this noise and structure confusion is that occasionally Nearest Sequence Memory exhibits "superstitious behavior" as also displayed by learning animals [Hilgard and Marquis, 1961; Pryor, 1984]. In certain circumstances, when the agent first finds success by some round-about route, it tends to use the route repeatedly. When exploratory steps uncover a more efficient path to reward, Nearest Sequence Memory will switch to the better path. For example, in the beginning of a particular learning run for the Space-Ship Docking task, the agent first docked successfully after making two redundant 180 degree turns. The agent then chose this behavior consistently, until, several trials later, random exploration discovered that the two extra turns were unnecessary, and the agent began docking without the extra "dance." This is over-specification—thinking more detail is relevant than really is. The extra "dance" is noise, but the agent interprets it as structure. Whether or not the Nearest Sequence Memory exhibits this behavior depends on how densely the state space around the reward is already covered by other instances when it first discovers the reward. This occasional tendency towards extra attention to details does imply that Nearest Sequence Memory could respond well to a teacher, because the agent would learn efficiently from a single presentation, and presumably, a teacher would demonstrate the task without the extra "dances."

Separating Instance-Based Learning and Dynamic Programming

As pointed out in the section on details of the algorithm, Nearest Sequence Memory performs two kinds of learning: instance-based population of the state space with raw experience and dynamic programming to calculate expected future reward. Currently these two kinds of learning are intertwined, but I believe the algorithm would perform better with a cleaner separation of the two. The instance-based component should be used to extract the relevant state space, then dynamic programming should be performed only on the extracted states.

The Utile Suffix Memory algorithm, introduced in the next chapter, uses this scheme.

Further Reducing Storage and Computation

By using the instance-limiting technique discussed in section 5.3, Nearest Sequence Memory is already computationally more efficient than its competitors. We may want to find ways of reducing the storage and computational requirements even further.

As an alternative to keeping the last N instances, the agent could also simply stop adding states to the chain once it thought it had sufficiently covered the possibly significant sequences. The instance chain would remain unchanged while the agent continued to learn the q-values for those instances.

Once learning is complete, we may want to compile the chain down into a smaller structure. Related work in [Stolcke and Omohundro, 1992] discusses a technique for compiling example sequences into hidden Markov models. Previous work discusses the application of this technique to building probabilistic finite state machines that include a notion of actions [McCallum, 1993a].

6 Efficient Utile Distinction Learning

Chapter Outline

This chapter presents Utile Suffix Memory, (USM), an algorithm that combines the advantages of the methods used in the previous two chapters. The algorithm uses instance-based learning, and thus learns quickly by making efficient use of previous experience; it also uses a utile distinction test, and thus makes task-relevant distinctions and handles noise well. Utile Suffix Memory is also closely related to previous work on suffix trees [Ron *et al.*, 1994]. The algorithm learns even faster than Nearest Sequence Memory.

The previous two chapters presented two algorithms with differing strengths and weaknesses. The first, Utile Distinction Memory, makes utile distinctions by using a statistical test that is robust to noise. However, it learns extremely slowly and has trouble building memories that are longer than one time-step.

These concerns are addressed by the second algorithm, Nearest Sequence Memory. This algorithm uses a technique termed "instance-based state identification" to handle experience efficiently, learn in much fewer steps and successfully build multi-step memories. NSM, specifically, is based on k-nearest neighbor.

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

What we want is the best of both algorithms. We want to combine instance-based learning with a utile distinction test. This chapter introduces Utile Suffix Memory, an algorithm that does just that.

6.1 Key Idea: Utile Distinctions with Instance-Based Learning

The key idea behind Utile Suffix Memory is the simple recognition that we can easily combine instance-based learning with a utile distinction test.

Nearest Sequence Memory and other k-nearest neighbor techniques use regional sample density to choose the size of the region over which they average instance data. In the case of Nearest Sequence Memory, this means that the length of memory that is considered significant is determined by how many times the agent had visited a certain situation before, not determined by any statistical measure of correlation between memory length and the ability to predict reward.

The use of a utile distinction test lets the agent explicitly determine how much memory is significant for predicting reward and how much isn't. It allows us to separate the reward variations due to noise from the reward variations that are due to task structure, and thus obtain the reward averages that include as much data as possible, without including data that really belongs to other perceptually aliased states. In other words, instead of grouping the data according to some arbitrary scheme, we explicitly partition the instances into groups that form a Markovian state space. Furthermore, the partitioning test is robust to noise.

What is thus far unspecified is the structure in which the agent maintains and tests these groups of instances. The algorithm presented in this chapter uses a type of finite state machine called a *suffix tree.* Note, however, that this key idea of combining utile distinctions with instance-based learning could use any scheme that allowed for explicit control of groupings or neighborhoods. For example, we could imagine applying these ideas to finite state machines, or schemes that used "soft" boundaries based on neighborhood functions with tunable weights [Stanfill and Waltz, 1986; Deng and Moore, 1995].

6.2 Utile Suffix Memory Algorithm

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

Like all instance-based algorithms, USM records each of its raw experiences. For a reinforcement learning agent, these are transitions consisting of action-percept-reward triples connected in a time-ordered chain. We will call these individual raw experiences *instances.* Unlike NSM, USM organizes, or "clusters," these instances in such a way as

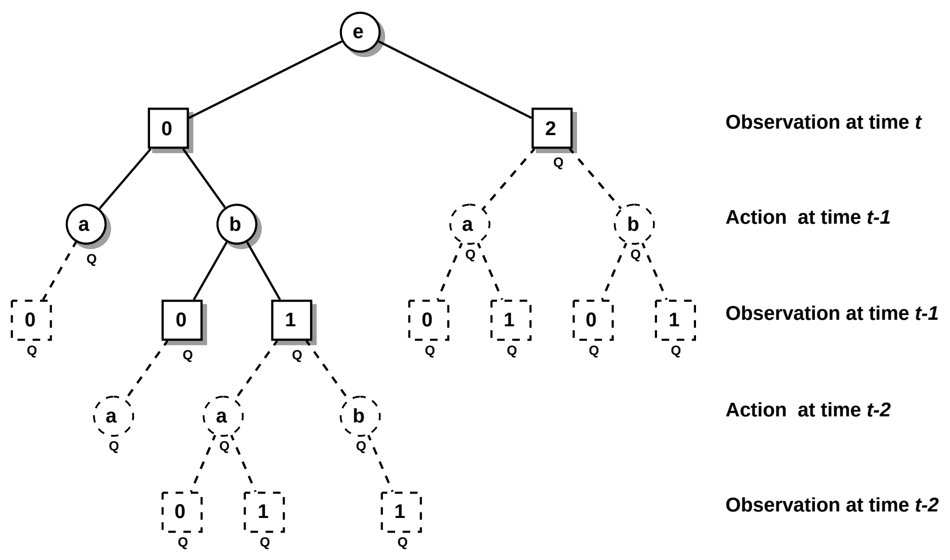

Figure 6.1: A USM-style Suffix Tree. Percepts are indicated by integers, actions by letters. The fringe nodes are drawn in dashed lines. Nodes labeled with a Q are nodes that hold Q-values.

to explicitly control how much of their history to consider significant. The structure that controls this clustering is a kind of finite state machine called a Prediction Suffix Tree [Ron *et al.*, 1994]. The structure can be thought of as an order-n Markov model, with varying n in different parts of state space.

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

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

The depth of the fringe is a configurable parameter. By using a multi-step fringe USM can successfully discover useful multi-step memories.

One way of looking at this algorithm is as a technique for incrementally building a tree-structured Q-table. The relationship between the work in this dissertation and other work based on tree-structured divisions of state space is discussed further in section 7.5.

6.3 Details of the Algorithm

The interaction between the agent and its environment is described by actions, observations and rewards. There is a finite set of possible actions, $\mathcal{A} = \{a_1, a_2, ..., a_{|\mathcal{A}|}\}$, a finite set of possible observations[1], $\mathcal{O} = \{o_1, o_2, ..., o_{|\mathcal{O}|}\}$, and scalar range of possible rewards, $\mathcal{R} = [x, y], x, y \in \Re$. At each time step, t, the agent executes an action, $a_t \in \mathcal{A}$, then as a result receives an observation, $o_{t+1} \in \mathcal{O}$, and a reward, $r_{t+1} \in \mathcal{R}$. We will use subscript i to indicate an arbitrary time.

Like other instance-based algorithms, Utile Suffix Memory records each of its raw experiences. The experience associated with time t is captured as a transition "instance" in four dimensional space, written T_t. The instance is a tuple consisting of all the available information associated with the transition to time t: the previous instance in the chain, the action, a_{t-1}, the resulting observation, o_t, and the resulting reward, r_t.

$$T_t = < T_{t-1}, a_{t-1}, o_t, r_t > \tag{6.1}$$

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

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

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

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

Below the "official" leaves of the suffix tree are additional layers of nodes called a fringe. Fringe nodes are labeled by the same scheme as the non-fringe nodes; they

[1]Section 8.2 discusses a natural extension for handling multi-dimensional continuous spaces.

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

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

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

Here are the steps of the USM algorithm:

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

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

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

$$T_t = < T_{t-1}, a_{t-1}, o_t, r_t > \tag{6.2}$$

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

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

$$\mathcal{T}(s) \leftarrow \mathcal{T}(s) \cup \{T_t\} \tag{6.3}$$

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

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

$$Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s'} \Pr(s'|s,a) U(s') \tag{6.4}$$

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

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

$$R(s,a) = \frac{\sum_{T_i \in \mathcal{T}(s,a)} r_i}{|\mathcal{T}(s,a)|} \tag{6.5}$$

$$\Pr(s'|s,a) = \frac{|\forall T_i \in \mathcal{T}(s,a) \ s.t. \ L(T_{i+1}) = s'|}{|\mathcal{T}(s,a)|} \tag{6.6}$$

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

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

implies that promoting the relevant fringe nodes into non-fringe leaf nodes will help the agent predict reward. Thus the agent only introduces new history distinctions when doing so will help the agent predict reward.

In keeping with the proof in section 4.2, we only compare the distributions of the leaf node's policy action and the fringe node's policy action.

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

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

$$Q(T_i) = r_i + \gamma U(L(T_{i+1})) \tag{6.7}$$

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

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

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

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

$$a_{t+1} = \text{argmax}_{a \in \mathcal{A}} Q(L(T_t), a) \tag{6.8}$$

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

Increment t and return to step 2.

6.4 Experimental Results

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

6.4.1 Hallway Navigation

The agent's task is to navigate to the goal location using actions for moving north, south, east and west. A key difficulty is that the robot's sensors provide only local information: binary indications the presence or absence of a barrier immediately adjacent to the robot in each of the four compass directions. Figure 6.2 shows a map of the environment; note that different actions are required in many of the perceptually aliased states. The agent receives reward 5.0 upon reaching the goal, reward -1.0 for attempting to move into a barrier, and reward -0.1 otherwise. After reaching the goal, the agent executes any action and begins a new trial in a randomly chosen corner of the world.

A graph of performance during learning is shown in figure 6.3. The agent used a temporal discount factor, $\gamma = 0.9$, a constant exploration probability $e = 0.1$, and a fringe of depth 3. USM learns quickly and handles noise well. Action noise, when added, consisted of executing a random action with probability 0.1, instead of the action chosen by the agent's policy; perceptual noise consisted of seeing a random observation with probability 0.1, and reward noise consisted of adding values uniformly chosen from the range -0.1 to $+0.1$ to the normal reward. For all combinations of noise, the agent learned the policy that would have been optimal if the action and perception noise did not interfere. In the graph, some suboptimal trial lengths continue to occur late in learning due to exploration steps, action noise and perception noise.

Many of the smarter exploration techniques, such as various counter-based techniques [Thrun, 1992b], which would have caused exploration to decay after an appropriate amount of experience, do not apply straightforwardly to tasks with hidden state. Efficient exploration with hidden state is an extremely difficult problem, and remains an area requiring work; Some of the problems regarding exploration with hidden state and a pointer to a solution are discussed further in section 8.2.7.

Figure 6.4 shows a suffix tree learned by Utile Suffix Memory. USM successfully separates the noise in the environment from the non-Markov structure in the environment, thus building only as much memory as needed to perform the task at hand, not as much as needed to model the entire environment, and definitely not as much as needed to represent uniformly-deep memories everywhere. Deep branches are created where needed to provide the multi-step memory required for disambiguating the various state 10's; shallower branches are created for the one-step memory needed to disambiguate the two state 5's on the critical path for this task, (but the deeper branches that would be needed to disambiguate the state 5's on the sides of the environment are not created because they are not needed to solve the task); and no memory is created at all for the states that are not aliased.

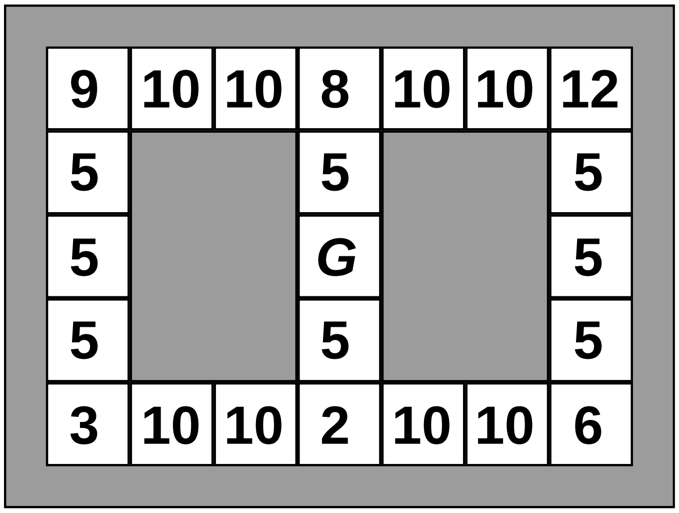

Figure 6.2: A hallway navigation task with limited sensors. World locations are labeled with integers that encode the four bits of perception.

6.4.2 Performance In Chrisman's Spaceship Docking Task

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

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

Utile Suffix memory also does not require more computation time. It is difficult to compare the time complexity of the algorithms directly, since they are based on different underlying constants, but the following analysis may give some insights. The Perceptual Distinctions Approach takes time $O(|\mathcal{S}|^2)$ per step to propagate forward the state occupation probabilities, plus $O(|\mathcal{S}||\mathcal{A}|)$ to calculate Q-values; in addition, every N steps, the agent performs the Baum Welch calculation, which requires time and space $O(|\mathcal{S}|^2|\mathcal{A}|N)$, and performs $(|\mathcal{S}|^2|\mathcal{A}| + |\mathcal{S}||\mathcal{O}|)$ Chi-Square tests; in Chrisman's

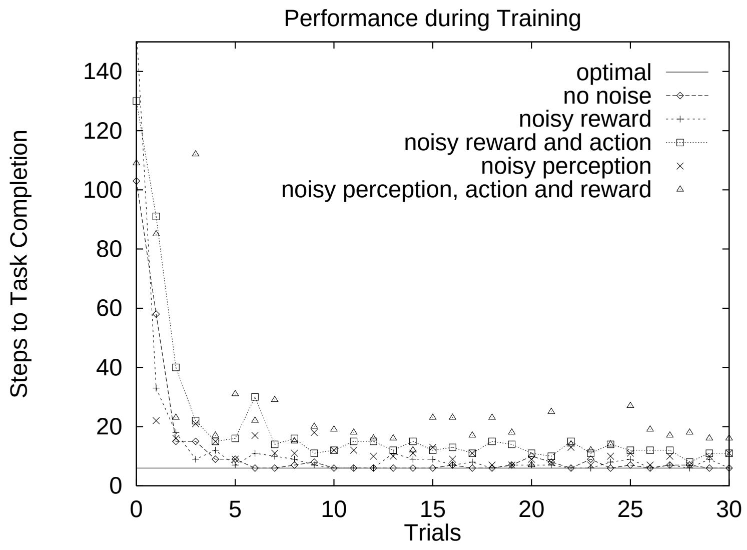

Figure 6.3: Performance during learning of the maze task shown in figure 6.2, with and without noise. The plot shows the median of ten runs.

experiments, N is 1000. Nearest Sequence Memory requires time $O(N)$ per step, where N is the number of instances in the chain, which in these experiments was limited to 1000. USM takes time $O(|\mathcal{S}|^2|\mathcal{A}|)$ per step if the agent uses full dynamic programming (but could take much less with Prioritized Sweeping), plus time for $O(K)$ Kolmogorov-Smirnov tests, where K is the number of fringe nodes under a single leaf node.

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

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

6.4.3 Whitehead's Blocks World Revisited

The agent's task is to pick and place red and blue blocks in order to uncover and lift the green block. The agent's available actions consist of *manipulative actions* for picking up and putting down blocks, as well as *perceptual actions*, in the form of visual routines [Ullman, 1984; Agre and Chapman, 1987], for directing its visual attention selectively. The visual-routine-manipulated region of attention is called a *marker*. The agent's perception consists of one bit that indicates whether the hand is empty or full, plus several bits that describe the features of the object being marked. Figure 6.6 depicts the task environment.

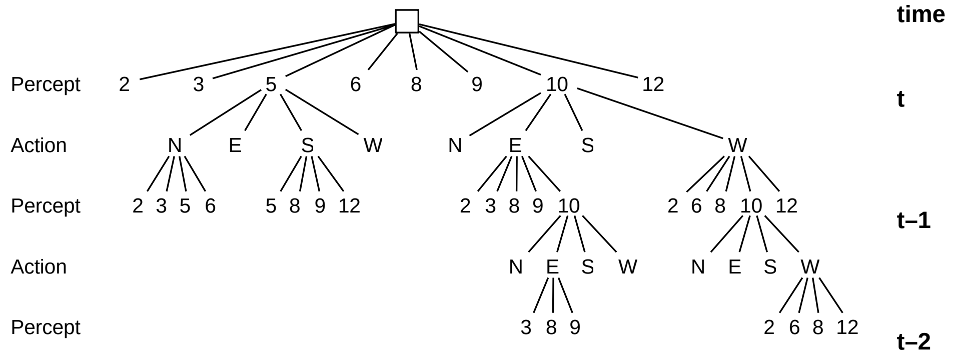

Figure 6.4: A tree learned by Utile Suffix Memory for navigating through the environment in the maze task shown in figure 6.2. Deeper branches of the tree correspond to longer, more detailed memories. USM created memory only where needed to solve the task at hand. The sequence 10-E-10-E-2 is missing because the agent never happened to have experienced that sequence.

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

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

USM was tested on this one-marker blocks task, using Whitehead's Learning by Watching technique, with a teacher intervention probability of 0.3. The teacher serves only to bias exploration, not to tell the agent which distinctions to make. Without the extra bias in this task, USM had trouble finding the sequences that lead to reward. Whitehead's *Lion* algorithm may have done better without a teacher because

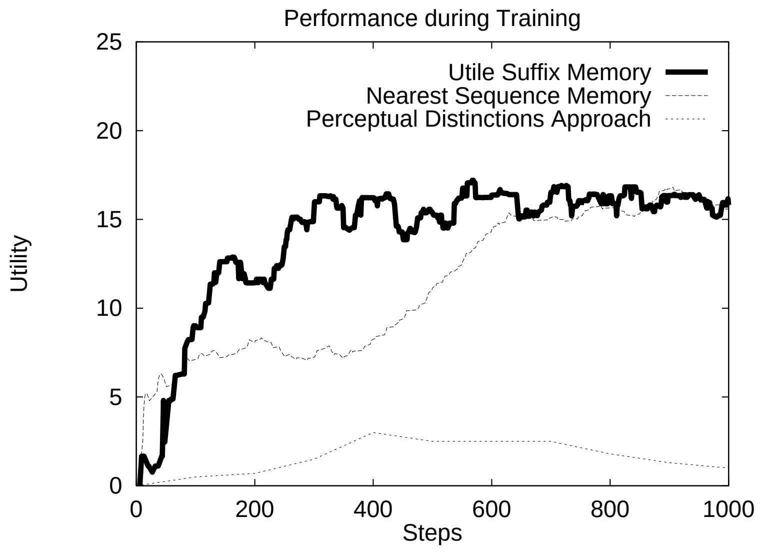

Figure 6.5: Performance during learning of Chrisman's spaceship docking task. The plot shows the median of ten runs.

(1) whenever it found a state to be aliased, it never visited it again, and (2) it used a strict schedule of perceptual and overt actions (four perceptual actions, then one overt action), whereas USM does not distinguish between overt and perceptual actions. As mentioned in section 6.4.1, finding smarter exploration for hidden state remains an area requiring much work. USM used a temporal discount factor, $\gamma = 0.9$, a constant exploration probability $e = 0.1$, and a fringe of depth 3. The tree that solves the task, built by USM after 15 trials, is shown in figure 6.6.

6.5 Discussion

Utile Suffix Memory has close ties to several other algorithms that use tree-structured partitions of state space. In particular, USM was inspired by Prediction Suffix Trees [Ron *et al.*, 1994], Parti-game [Moore, 1993], G-algorithm [Chapman and Kaelbling, 1991], and Variable Resolution Dynamic Programming [Moore, 1991]. Relations to these algorithms are discussed in more detail in section 7.5.

6.5.1 Reason for Better Performance

Utile Suffix Memory displays improved performance—even beyond that of Nearest Sequence Memory. The improvement results from combining the advantages of instance-based learning and utile distinctions. USM records all its raw data for easy reinterpretation whenever state space boundaries shift; it also uses robust statistical tests to set those boundaries as coarsely as possible while still separating regions of different state-action utility. UDM offers utile distinctions based on a statistical tests that are robust to noise, but it requires re-gathering experience after each state space split. NSM saves

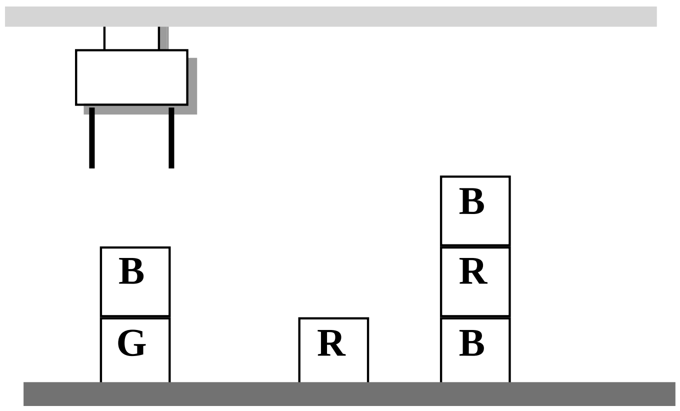

Figure 6.6: A hand-eye coordination task with blocks, using visual routines. The agent must uncover the green block and pick it up.

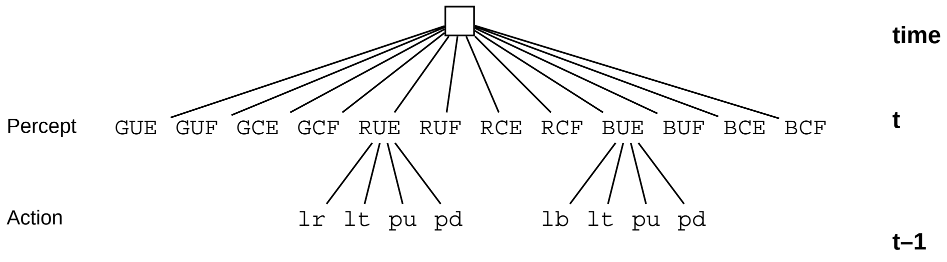

Figure 6.7: The suffix tree learned by USM for the blocks task. R=red, G=green, B=blue, U=uncovered, C=covered, E=hand-empty, F=hand-full; lr=look-red, lg=look-green, lb=look-blue, lt=look-table, ls=look-top-of-stack, pu=pick-up, pd=put-down.

all its raw experience, but it does not partition state space so as to agglomerate as much experience as was relevant.

Another reason for improvement over NSM is that USM learns a state transition model and thus can use dynamic programming directly on this model. This allows us to separate model learning, which requires exploration, for the "non-learning" dynamic programming, which only requires computation. For instance, with such a scheme, reward can be easily propagated from crucial regions of state space that are, nevertheless, difficult to visit multiple times. In Q-learning–based algorithms, reward propagation requires multiple runs through the propagation path. Even in path play-back schemes like Experience Replay [Lin, 1991a], the agent must still visit the reward state multiple times in order to propagate reward out along the multiple different paths to the goal that may be available. With USM, paths that have previously been explored, but have not previously lead to the goal, can still be used to propagate reward. In addition to USM and VRDP, several other reinforcement learning algorithms take advantage of a model-based approach [Sutton, 1990a; Barto *et al.*, 1991; Moore and Atkeson, 1993]. This idea is also discussed in section 7.5.4.

Another reason for improvement over UDM is that USM uses a state space representation that is easier to learn than that of UDM. UDM tries to learn a POMDP—a state model with loops, which provide infinitely long memories. USM, however, learns a finite length memory in the form of linear suffixes. USM thus searches a much simpler space of model possibilities. There are many cases in which models without loops are adequate. Note, however, that the instance-based learning aspect of USM is not tied to this finite-suffix model; we could easily imagine some Baum-Welch-like POMDP learner that also took advantage of instance-based learning.

6.5.2 Areas for Improvement

The most limiting weakness of Utile Suffix Memory is that it cannot efficiently handle tasks with large sensory spaces. Since the branching factor of the tree is the same as the size of the number of possible observations, size of the tree would grow prohibitively fast when there is more than a trivial number of possible percepts. (Note that the experimental results all used tasks with a $|\mathcal{O}| < 20$.) USM cannot select individual dimensions of perception. Addressing this concern is the subject of the next chapter.

There are other issues that merit discussion, but I will address these at the end of the next chapter.

7 Utile Distinctions for Perception and Memory

Chapter Outline

This chapter presents U-Tree, an algorithm that not only selects utile *memory* distinctions, but also utile *perceptual* distinctions. This means that the agent can choose to attend to certain perceptual features, and ignore others—and thus we have, for the first time, an algorithm that can handle large perceptual spaces. U-Tree is based on Utile Suffix Memory.

Experimental results are shown for a complex driving task with utile hidden state, over 21,000 world states and 2,500 percepts. The agent learns a task-dependent state space that contains about 150 states.

This is the last chapter in the series of four chapters that present new algorithms.

Utile Suffix Memory, the algorithm presented in the previous chapter, performs well, but it suffers from an important limitation: it cannot handle large perceptual spaces well. The branching factor of the suffix tree is as large as the size of the sensor space. The nested branches cause an exponential explosion in the number of nodes, and if the algorithm is applied to a task with many sensor states, and the tree quickly grows beyond a manageable size.

In this chapter, I present a new algorithm, called U-Tree, that addresses this problem by learning a *Factored State Representation* (or *Structured State Representation*).

7.1 Key Idea: Selecting Memory and Perceptual Distinctions

The key idea behind U-Tree is that the same tree structure used to represent state distinctions based on memory, can also be used to represent state distinctions based on features of perception.

The algorithm described in the previous chapter, Utile Suffix Memory, treats a percept as an indivisible whole, and thus, it has tree nodes with as many branches as there are percepts. By treating a percept as multi-dimensional vector of features, and by branching based only on one feature in the vector at a time, tree nodes can instead have only as many branches as there are values for that individual feature. Obviously, the total number of percepts has not changed; but we have given the agent the ability to ignore certain dimensions of perception, and thus the ability to build an internal state space that is much smaller than the space of all possible percepts.

What is meant by dimension of perception? Perception can naturally fall into different categories or features, like color, distance and direction. For instance, in the Blocks World example in section 6.4.3, the emptiness or fullness of the hand is one dimension, the number of blocks above the gaze point is another dimension, the color at the gaze point a third. In this new algorithm, when the agent adds a distinction, it can select just one dimension of perception, and the branching factor is only as large as the size of that dimension. For example, the branching factor of the hand-fullness dimension would be 2.

These individual features may be bits, as in the G-algorithm [Chapman, 1989], or may be multi-valued features, as used in the driving domain described in section 7.4.1. Note that in addition to handling discrete perception, this perceptual distinction scheme could also handle continuous spaces. The branches of a tree node add distinctions that divide the continuous space along certain boundaries. The Parti-game algorithm [Moore, 1993] is an example of a tree-based algorithm that divides a continuous space. The G-algorithm, Parti-game, and other related work with trees are discussed in section 7.5. Ideas about applying U-Tree to continuous spaces are discussed in section 8.2.2.

The particularly beneficial aspect of combining perceptual distinction trees with suffix trees is that this allows the agent to represent both perceptual and memory distinctions in the same uniform structure—recognizing the inherent similarity between the two, as discussed in section 1.1. We combine perceptual distinctions and memory distinctions in one big pot of possibilities, and employ the same statistical techniques as used in the last chapter to extract those distinctions that are utile.

This algorithm thus demonstrates by example the tight link between hidden state and feature selection. In both cases, a *Factored State Representation* can capture just those state distinctions that are necessary. The algorithm can also be understood as doing *Value Function Approximation* in that it builds a representation of the value function using a more compact structure than a mapping from all world states to return values.

After a long journey—beginning in chapter 1 with comments about the tight interrelation between hidden state and selective perception, and continuing through three chapters containing three algorithms—we have finally arrived at a technique that can handle both hidden state and selective perception. We have arrived at a technique that can handle both "too much sensory data" and "too little sensory data."

7.2 U-Tree Algorithm

The U-Tree algorithm is closely related to the algorithm described in the previous chapter, Utile Suffix Memory. Like Utile Suffix Memory, it is instance-based: U-Tree groups the instances in the leaves of a tree; and it uses statistical tests on the fringe in order to make utile distinctions. The difference is that the agent can explicitly control which features of perception are used.

In USM the agent begins with as many leaves (states) as there are percepts. In U-Tree, the agent has the possibility of having many fewer states than percepts by picking out only those features and conjunctions of features that show statistical significance in predicting reward. U-Tree begins with truly *no distinctions*; all percepts are aliased to one internal state—the single node root of the tree. With experience, the agent finds utile distinctions and selectively adds branches to the tree.

As in USM, a leaf of the tree acts as a bucket, holding a cluster of instances that share certain features in common. The features associated with a certain leaf are determined by the nodes on the path from that leaf to the root of the tree. Each interior node of the tree introduces a new distinction. Unlike USM, the distinctions are based not just on the depth of the branch, but two parameters: (1) a "perceptual dimension," indicating which individual feature (or dimension of perception) will be examined in order to determine which of the node's branches an instance should fall down, and (2) a "history index," indicating at how many time steps backward from the current time this feature will be examined. By using a non-zero history index, tree branches can distinguish between instances based on past features, and can thus represent short-term memory.

Figure 7.1 depicts a tree and instance chain based on a simple abstract task in which the agent can execute two actions (labeled 'u' and 'v'), and receives observations comprised of three features (labeled 'Dimension 1', '2', and '3'). For example, in a simple navigation task, actions 'u' and 'v' may correspond to turning left or right; 'Dimension 1' may correspond to a range sensor facing right, with 'A' indicating 'one foot', 'B' indicating 'three feet' and 'C' indicating 'far'; 'Dimension 2' may correspond to a range sensor facing left, and 'Dimension 3' may correspond to a bump sensor, with '+' indicating touching and '−' indicating open.

Consider the moment at which the agent has just experienced the sequence leading up to the starred instance, (observation 'B$−'). In order to find the leaf node in which the instance belongs, we begin at the root of the tree, and see that it has branches labeled '0@', '0#' and '0$'. Thus it adds a distinction based on the value of 'Dimension 2' at zero time steps backward. Since our instance has a '$' for 'Dimension 2', we fall down the right-hand branch. The node at the bottom of the right-hand branch has child branches labeled '1u' and '1v', and thus adds a distinction based on the action taken one time step ago. Since, one time step ago, the agent executed action 'u', (the transition coming into the starred instance is labeled with action 'u'), we fall down the left-hand branch. The node at the bottom of that left-hand branch has child branches labeled '1A', '1B' and '1C', and thus adds a distinction based on the value of 'Dimension 1' one time step ago. Since, one time step ago, the agent observation was 'C$+', (that is the

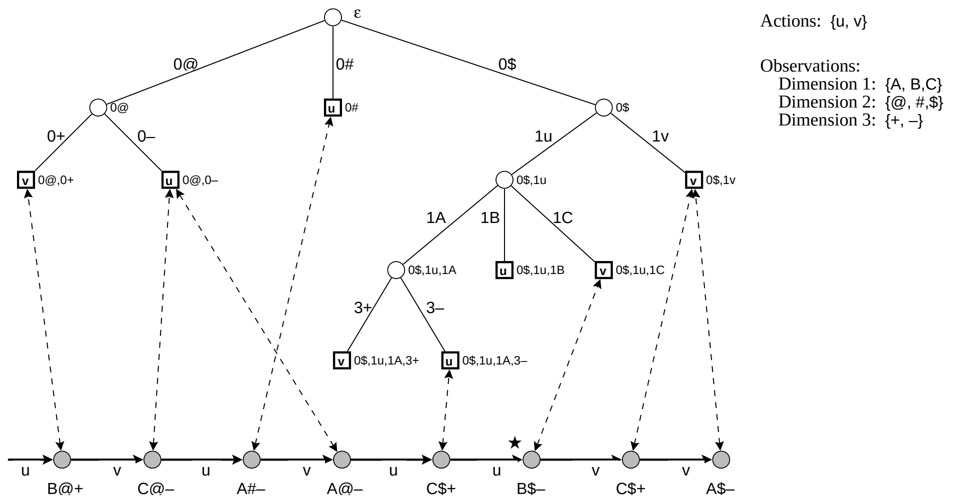

Figure 7.1: An example of a U-Tree instance-chain and tree, showing the relationship between instances (gray circles) and leaves (squares). The instance-chain is at the bottom; each instance represents a step in the world; every circle is labeled by the observation received at that time step, and every arrow is labeled by the action taken to make that transition. In the tree, each branch is labeled by its history index and perceptual dimension; beside each tree node is the conjunction of features the node represents. The tree nodes drawn as squares are the agent's internal states; each contains the Q-values for each action, although the figure only shows the policy action. The dashed-arrows show which which agent internal state holds each instance.

label on the instance to the left of the starred instance), we fall down the right-hand branch, which is labeled '1C'. Here we are at a leaf, and we deposit the instance. Inside that leaf the agent will also find the policy action to be executed—in this case 'v'. The dashed line shows the association between the instance and the leaf.

While USM's trees had only one possible way to grow, these trees have many. In USM, each branch distinguishes the current state based on all dimensions of perception, and one deeper layer of the tree always corresponds to one step back in time; in U-Tree there is a choice of which feature this branch will distinguish, and a branch can consider 0 or several time steps backwards from its parent. Note that the agent can now represent jumps or gaps in the history window.

Having many possible ways to grow the fringe makes fringe generation harder than it was before. However, there are tricks to reduce the size of the fringe; these are described in the next section. With USM it was conceivable to cache the fringe from one split test to the next; here that is no longer so easy.

There is no escape from the curse of dimensionality. The *curse of dimensionality* refers to the way in which state spaces increase in size exponentially with respect to the number of input dimensions. U-Tree attempts to hide large state spaces from the policy, (and this has distinct advantages), but the problem has not gone away—the agent simply struggles with it at a different level. There are several techniques used to help reduce the number of fringe expansions the agent tests, and thus better manage the curse of dimensionality. Some of these techniques are "lossy," others "non-lossy." By "lossy" I mean techniques that disallow certain splits that mights otherwise have happened. Examples of "non-lossy" fringe reduction techniques are: prune a branch whenever there no transitions in that branch; prune a branch whenever there is no deviation in the statistics for that branch. An example of "lossy" fringe reduction technique is: enforce a fixed ranking on the order with which fringe distinctions are nested; (this biases the order in which distinctions are nested in the tree). These techniques are discussed in more detail in the next section.

7.3 Details of the Algorithm

The details of this algorithm have much in common with Utile Suffix Memory. Instead of including only the differences, however, and thereby forcing the reader to remember the details of an algorithm presented in the previous chapter, I include all the details here—repeating them where necessary.

The interaction between the agent and its environment is described by actions, rewards, and observations. There is a finite set of possible actions, $\mathcal{A} = \{a_1, a_2, ..., a_{|\mathcal{A}|}\}$, scalar range of possible rewards, $\mathcal{R} = [x, y], x, y \in \Re$, a finite set of possible observations, $\mathcal{O} = \{o_1, o_2, ..., o_{|\mathcal{O}|}\}$. At each time step, t, the agent executes an action, $a_t \in \mathcal{A}$, then as a result receives an observation, $o_{t+1} \in \mathcal{O}$, and a reward, $r_{t+1} \in \mathcal{R}$. We will use subscript i to indicate an arbitrary time.

The set of observations is comprised of the set of all possible values of a perceptual vector. The vector has a finite number, m, of dimensions, or "perceptual features," $\mathcal{D} = \{D_1, D_2, ..., D_m\}$. We use subscript d to indicate an arbitrary dimension index. Each feature is an element of a finite set of possible values, $D_d = \{D_{d,1}, D_{d,2}, ..., D_{d,|D_d|}\}$. The value of dimension D_d of the observation at time t is written: $o[d]_t$. Thus, we write

$$o_t = \langle o[1]_t, o[2]_t, ..., o[m]_t \rangle \,. \tag{7.1}$$

The size of $\mathcal{O}$ can be calculated from the sizes of the dimensions that comprise it:

$$|\mathcal{O}| = \prod_{d=1}^{|\mathcal{D}|} |D_d|. \tag{7.2}$$

Like other instance-based algorithms, U-Tree records each of its raw experiences. The experience associated with time t is captured as a transition "instance" in four dimensional space, written T_t. The instance is a tuple consisting of all the available information associated with the transition to time t: the previous instance in the chain, the action, a_{t-1}, the resulting observation, o_t, and the resulting reward, r_t:

$$T_t = \langle T_{t-1}, a_{t-1}, o_t, r_t \rangle \,. \tag{7.3}$$

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

In addition to organizing the instances in a time-ordered chain, U-Tree also clusters the instances in the nodes of a tree.

Tree nodes are labeled in such a way that each node adds a distinction based on the combination of two parameters: one, a "history index," indicating a certain number of steps backward in time, written h; and two, a dimension of perception, or the action dimension, written d. We include the set of actions among the the perceptual feature sets so as to allow the agent to add a distinction based on a previous action, (as in Utile Suffix Memory).

The history index of node n is written n_h; the dimension of node n is written n_d. A node, n, has as many children as $|D_{n_d}|$. No ancestor of a node, n, has the same history index and dimension label as n. Each node of the tree can thus be uniquely identified by the set of labels on the path from the node to the root. This set, s, is called the node's *conjunction*. We will use s interchangeably to indicate the conjunction as well as the tree node that the conjunction specifies.

An instance, T, is deposited in the leaf node whose conjunction, s, is satisfied by the actions and observations of the transition instances that precede T in time. The set of instances associated with the leaf labeled s is written $\mathcal{T}(s)$. The tree leaf to which instance T belongs is written $L(T)$.

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

The leaves of the suffix tree are the internal states of the reinforcement learning agent. Thus, deep branches of the tree correspond to regions of the agent's internal state space that are "finely distinguished" by the conjunction of many features, possibly including long memories. Shallow branches of the tree correspond to regions of the agent's internal state space that are only "broadly distinguished", with few distinctions made. The agent maintains learned estimates of expected future discounted reward for each state-action pair. The estimate, or Q-value, for choosing action a from leaf with suffix s is written $Q(s, a)$. Note that, although deeper layers of the tree correspond to earlier time steps, all Q-values indicate the expected values for the next step in the future.

In Utile Suffix Memory there was only one way to expand the fringe, and thus is was reasonable to cache the fringe from one step to the next. Here there are many ways to expand the fringe, and thus the agent will expand it and destroy it each time it wants to test for new distinctions.

Here are the steps of the U-Tree algorithm:

1. The agent begins with a tree that represents no distinctions. This tree has only one node—a root node.

   For this root node s, $\mathcal{T}(s)$ is empty. The time-ordered chain of instances is also empty.

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

   $$T_t = < T_{t-1}, a_{t-1}, o_t, r_t > . \tag{7.4}$$

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

   The agent also associates the new instance with the leaf node whose conjunction is satisfied by this instance. We find this node by the same method used to place an exemplar in a decision tree. We start at the root of the tree, and fall down the branch corresponding to the feature value exhibited by this instance. We continue down the tree until a leaf is reached. For this leaf, s:

   $$\mathcal{T}(s) \leftarrow \mathcal{T}(s) \cup \{T_t\}. \tag{7.5}$$

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

   Value iteration consists of performing one step of dynamic programming on the Q-values:
$$Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s'} \Pr(s'|s,a)U(s') \tag{7.6}$$
   where $R(s,a)$ is the estimated immediate reward for executing action a from state s, $\Pr(s'|s,a)$ is the estimated probability that the agent arrives in state s' given that it executed action a from state s, and $U(s')$ is the utility of state s', calculated as $U(s') = \max_{a\in\mathcal{A}} Q(s',a)$.

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

$$R(s,a) = \frac{\sum_{T_i\in\mathcal{T}(s,a)} r_i}{|\mathcal{T}(s,a)|} \tag{7.7}$$
$$\Pr(s'|s,a) = \frac{|\forall T_i \in \mathcal{T}(s,a) \ s.t.\ L(T_{i+1}) = s'|}{|\mathcal{T}(s,a)|} \tag{7.8}$$

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

4. After every k steps, the agent tests whether newly added information or dynamic programming has changed the transition utility statistics enough to warrant adding any new distinctions to the agent's internal state space. The agent expands the fringe, and uses the Kolmogorov-Smirnov test to compare the distributions of future discounted reward associated with the same action from different nodes. If the test indicates that two distributions have a statistically significant difference, this implies that promoting the relevant fringe nodes into non-fringe leaf nodes will help the agent predict reward.

   There are many possible variations on techniques for expanding the fringe. The most straightforward is to expand the fringe by all possible permutations of observations and actions to a fixed depth, z, using a maximum history index of h. This however, results in
$$(h(|\mathcal{D}|+1))^z \tag{7.9}$$

possible expansions for each leaf, unless some pruning occurs.

The number of expansions can be reduced considerably through a number of techniques. Here are some of the techniques used by the current implementation:

- Many leaves may contain zero or few instances. There is no need to expand the fringe below those leaves.
- Many leaves may contain many instances, but have little deviation between those instances' utility values. There is no need to expand the fringe below those leaves.
- The order of the terms in a conjunction don't matter, yet the straightforward expansion tests different orderings of the same conjunction. We choose to test only one ordering by defining a ranking on the perceptual dimensions, followed by the action dimension, and only making expansions that conform to the ranking. Note that this strongly biases the nesting order of distinctions.

All these techniques help considerably, but even so, testing the fringe is a significant computational load. Section 8.2.10 discusses some ideas for remedying this situation.

Utile Suffix Memory tested the fringe after every step, (*i.e.* $k = 1$). The parameter k is configurable—trading off the timeliness of making new possible splits against the computational effort required for each round of testing. Since U-Tree requires much more computation per fringe expansion, I have used values of k between 100 and 1000 in experiments.

In keeping with the proof in section 4.2, the agent only tests the distributions of the leaf node's policy action and the fringe node's policy action.

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

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

$$Q(T_i) = r_i + \gamma \sum_{L(T_{i+1})} \Pr(L(T_{i+1}))U(L(T_{i+1})) \tag{7.10}$$

where $\Pr(L(T_{i+1}))$ is calculated using equation 7.8.

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

Note that, when a fringe node that is several fringe-layers deep passes the utile distinction test, not only is that fringe node made into an official node of the tree, but also that node's uncles and great-uncles.

When a split is made, the agent starts the split test again to look for further possible splits that the new distinction may bring to light.

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

$$a_{t+1} = \mathrm{argmax}_{a \in \mathcal{A}} Q(L(T_t), a) \tag{7.11}$$

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

Increment t and return to step 2.

7.4 Experimental Results

The section presents experimental results in a difficult road driving task, with much utile non-Markov hidden state, and a large perception space.

7.4.1 New York Driving

The agent's task is to weave in and out of one-way traffic, using actions and perceptions based on visual routines. The agent shares the road with both faster and slower cars. The agent has control of its steering, but no control of its speed. I call the task "New York Driving."

The environment differs from the tasks shown previously in that it has a much larger world state space and sensor state space—over 21,000 world states and over 2,500 sensor states; and there is also much utile non-Markov hidden state. Performance on this task strongly benefits from the combinations of both selective perception and short-term memory.

The Environment

The environment consists of four lanes of uni-directional traffic. The traffic includes the agent's car and many "obstacle" cars. The agent's car makes forward progress at a constant rate of 16 meters per second. Some obstacle cars travel more slowly, moving at 12 meters per second; we call these "slow cars." Other obstacle cars travel more quickly than the agent, moving at 20 meters per second; we call these "fast cars." The agent car has the ability to change lanes; obstacle cars do not change lanes.

When the agent car runs into the back of a slow car in its lane, the two cars do a "NYC squeeze," scraping each other's sides, but still managing to squeeze by each other within that single lane. When a fast car reaches the back of a slow car, the fast

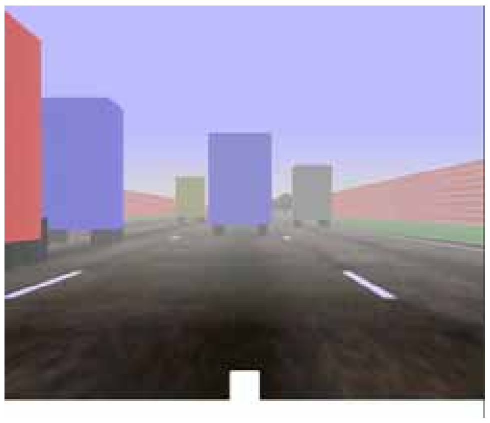

Figure 7.2: The perception from the driver's point of view, showing the driver's hood, hood ornament, the highway, and trucks.

car slows down to match the speed of the slow car. When a fast car reaches the back of the agent's car, the fast car slows down to match the speed of the agent's car, and also begins beeping its horn, continuing to beep until the agent gets out of the way by shifting into another lane; at that point the fast car continues at 20 meters per second.

Time is discrete at the resolution of one-half second per step. At each step all cars instantaneously change position according to their speed. During a time step in which the agent car changes lanes, it both shifts lanes and moves forward its normal distance.

New slow cars appear in randomly selected lanes on the agent's forward horizon; new fast cars appear in randomly selected lanes on the agent's rear horizon. At each half-second a new car appears with probability 0.5. The new car is chosen to be fast or slow with equal probability.

Slow cars are more likely to appear in the right-hand lanes; fast cars are more likely to appear in the left-hand lanes. When a slow car is added, it appears in the right-most lane (lane 4) with probability 0.4, it appears in the next-to-right-most lane (lane 3) with probability 0.3, appears in lane 2 with probability 0.2, and appears in the left-most lane (lane 1) with probability 0.1. Fast cars are added using the mirror image of the slow cars distribution; when a fast car is added it appears in lane 1 with probability 0.4, lane 2 with probability 0.3, lane 3 with probability 0.2 and lane 4 with probability 0.1. (In those trials in which the obstacle car speed parameters were changed such that only one speed of obstacle car was used, the cars were uniformly distributed across all lanes.)

Obstacle cars have a color, randomly chosen from red, blue, yellow, white, tan, and gray. The road and shoulder have a color randomly chosen from yellow, white, tan and gray.

The agent's visual horizon is 66 meters meters in front and behind of the agent's view-point in the car. The length of road around the agent car can be seen as seventeen discrete four-meter-long sections—eight in front of the agent's car and eight behind the agent's car, and one occupied by the agent's car.

Figure 7.2 shows a scene from the agent driver's point of view.

The reward function delivers one of three possible rewards at each time step. Whenever the agent scrapes by a slower car, it receives a negative reward of -10.0; whenever a fast car is honking its horn at the agent, the agent receives a negative reward of -1.0; otherwise, the agent receives positive reward of 0.1 for making clear forward progress.

The agent navigates using five actions, which are based on visual routines [Ullman, 1984; Agre and Chapman, 1987]. These actions are: gaze-forward-left, gaze-forward-center, gaze-forward-right, gaze-backward, and shift-to-gaze-lane. Table 7.1 also lists the actions. The gaze-forward visual routines begin by positioning the agent's gaze immediately in front of the agent in the left, center or right lane (relative to the agent's lane), then the routines trace the gaze forward along that lane until either an obstacle car or the horizon is reached. The gaze-backward action performs the same lane-tracing behavior, except it begins immediately behind the agent and traces backward; the lane in which the agent traces backwards is the same lane (left, center or right) in which the agent was looking previously. (Thus, for example, there is no way to shift gaze from forward right to backwards left with a single action. The agent must execute gaze-forward-left, then gaze-backward.) The shift-to-gaze-lane action is a deictic action that causes the agent's car to shift into the lane at which the agent is currently looking. This action works whether the agent is looking forward or backward. As the last step of executing the shift-to-gaze-lane action, the agent's gaze is placed at the forward center of its new lane, as if it had executed an implicit gaze-forward-center.

Action	Description
gaze-forward-left	Look at closest car in lane to the left.
gaze-forward-center	Look at closest car in same lane as agent.
gaze-forward-right	Look at closest car in lane to the right.
gaze-backward	Look backwards at closest car in current gaze lane.
shift-to-gaze-lane	Steer car into lane where agent is looking.

Table 7.1: The actions of the agent driver

The agent's sensory system delivers seven dimensions of sensor information; these are: hear-horn, gaze-object, gaze-side, gaze-direction, gaze-speed, gaze-distance, gaze-refined-distance, and gaze-color. These dimensions of perception are also shown in table 7.2. Hear-horn is one of two values, indicating whether or not a fast car is on the agent's tail, beeping its horn. Gaze-object indicates whether the agent's gaze is pointed at a car, the road or the shoulder. The only situations in which the agent's gaze will fall on the road is when it looks into a lane with no cars in the specified direction. Gaze-side indicates whether the agent's gaze is left, center or right of the agent car position. Gaze-direction indicates forward or backward. Gaze-speed indicates whether the object the agent is looking at is looming or receding. When looking forward, the road and shoulder are always looming; slow cars also loom; fast cars recede. Gaze distance gives a rough-grained indication of the distance to the gaze point, indicating far, near or "nose," which is closer than near. "Nose" covers distances from 0 to 8 meters, "near" distances from 8 meters to 12 meters, and "far" all the further distances. Gaze-refined-distance provides the agent with a fine-grained distance measure, by dividing each of the gaze-distance regions into two. Because it is in a separate "dimension," the agent can choose to have finer grain resolution in some situations, but only coarse-grained resolution in others. Gaze-color is one of six values—either red, blue, yellow, white, gray or tan if looking at a car, and yellow, white, gray or tan if looking at the road or shoulder.

Dimension	size	values
Hear horn	2	yes, no
Gaze object	3	car, shoulder, road
Gaze side	3	left, center, right
Gaze direction	2	forward, backward
Gaze speed	2	looming, receding
Gaze distance	3	far, near, nose
Gaze refined distance	2	far-half, near-half
Gaze color	6	red, blue, yellow, white, gray, tan

Table 7.2: Sensory system of the agent driver

What is the size of the sensory state space? With two possible horn states, three possible gaze objects, two possible gaze speeds, three possible sides, two directions, three distances, two refined distances and six colors, there are 2,592 possible sensor states. Note, however, that some of these sensor states never occur in this environment: whenever looking at the shoulder or the road: (1) the gaze distance is always at the horizon, (2) the speed is always exactly correlated with the gaze direction and (3) the gaze color is one of gray, white or yellow. Thus, factoring out the distance dimensions, the speed and the color for the road and shoulder gaze objects, there are actually 648 sensor states experienced by the agent. Although, note that, in general, the agent has no way of knowing this ahead of time; traditional Q-learning would still have created a Q-table of size 2,592, (which is not overwhelming large in this case, but still would have both (1) missed a significant opportunity for generalization and (2) given the agent a hidden state problem).

What is the size of the world state space? There are four lanes, and seventeen four-meter length positions for obstacle cars, thirteen possible occupations of each four-meter length of each lane (empty, fast car, slow car, and six possible colors for each car), and four possible lanes for the agent car. The direction of the agent's sensors is also part of world state. The sensors can be pointing in three possible lanes relative to the agent, two possible directions (forward and back). All the other sensor dimensions are determined by the state of the exterior world in conjunction with the sensor direction. Without including the direction of the agent's sensor system, there are 3536 possible world states. Including the configuration of the agent's sensor system, there are 21,216 world states.

```
PrevAction             : [ 4] Goto gaze-lane
Perception:
 [0] Gaze color        : [ 0] Red
 [1] Gaze refined dist : [ 0] Far-half
 [2] Gaze distance     : [ 2] Far
 [3] Gaze speed        : [ 0] Looming
 [4] Gaze direction    : [ 0] Forward
 [5] Gaze side         : [ 1] Center
 [6] Gaze object       : [ 1] Car
 [7] Hear horn         : [ 0] No

   0 1 2 3 4 5
16 ;         ;
15 |   R<    |
14 '         '
13 ;     G   ;
12 |         |
11 '         '
10 ;       W ;
 9 |         |
 8 '   O     '
 7 ;     R   ;
 6 |       G |
 5 ' w       '
 4 ;         ;
 3 |         |
 2 ' b   B   '
 1 ;         ;
 0 |         |
```

Figure 7.3: A picture of the driver's perceivable environment. The '`O`' indicates the agent driver's car; the other capital letters indicate slower obstacles cars; the lowercase letters indicate faster obstacle cars. The '`<`' character points at the driver's gaze point. The shoulder is drawn with '`|`' and '`;`' characters. From this position, the driver next looked right, shifted right, then looked right and shifted right again—squeezing between the gray ('`G`') and white ('`W`') slower cars.

The Task, Hidden State and Selective Perception

Good performance on the task involves avoiding slower cars and avoiding getting beeped at by faster cars.

To avoid slower cars, a driver must gaze forward to check for looming cars in its lane, and when it sees a looming car, search the adjacent lanes for the most clear alternative lane.

To avoid getting beeped at for long, the driver will also shift lanes to get out of the way of tail-gating faster cars. This involves the same kind of lane search. Highly skilled drivers may even gaze backwards occasionally to shift out of the way of looming faster cars even before they hear the horn once.

This task involves interacting with a changing world and responding to severe time pressure. Many other tasks used as reinforcement learning test-beds, such as box pushing, block stacking, mazes, etc, do not have this difficulty.

Note that there is utile non-Markov hidden state involved in the search for alternative lanes. In some situations seeing a car in the left lane means that the driver should look right to find a more clear lane, but if the driver has already looked right, and the right lane contains a looming car that is even closer than the looming car in the left lane, the agent should shift left anyway, (and hope to have time to safely shift again later). This is one example of hidden state, there are many others, some of which we will discuss in more detail in section 7.4.1.

Note that selective perception is also beneficial in this task because not all sensory dimensions are relevant in all situations. The most blatant case is that the distance and speed dimensions are not significant when seeing the road or the shoulder, because those dimensions always have the same value when looking at the road or shoulder. There are also more subtle examples of task irrelevant features, especially when considered in conjunction with short-term memory. For instance, when an agent looks behind to avoid getting beeped at, if it sees a receding (slower) car, it does not matter how far away that car is.

Trying to solve this task with an algorithm that makes perceptual distinctions, as opposed to only utile distinctions, would be a horrible mistake. This strategy would require an extremely large number of distinctions. For example, it would involve building memories for predicting that, if we looked right, looked left, and then looked right again we would probably see the same color car. The agent would create a huge state space for making a plenitude of these predictions that are totally unnecessary for solving the task. Thus, for instance, there is no way that we could use the Perceptual Distinctions Approach to solve this task—not just because of the inefficient use of experience and long training time, but because learning perceptual distinctions here would create a state space that was far too big.

Performance

The section contains an outline of current performance results.

Comparing Performance Against a Human-Written Policy in an Environment With Only Slower Cars

In order to obtain concrete experience with how difficult it is to solve this task, first I wrote a policy myself. In order to be fair, I coded a policy with much the same structure as the tree used by U-Tree. The policy consists of nested if-then-else statements, each of which looks at one or more dimensions of perception, and has the ability to look backwards to a particular history index. I found this structure quite intuitive; much more intuitive than filling in a Q-table, for instance. Note that the if-then-else structure has the ability to be quite a bit more compact than the agent's tree since several values for one perception dimension can be combined into one branch by using the "or" operation. Section 8.2.8 discusses this idea further.

In a little over three hours I wrote a policy that I thought would do well when there were only slower cars, but no faster cars. The policy's if-then-else tree has 32 leaves. I found that thinking about all the additions necessary for handling faster cars was too tedious, so I stopped there.

I modified the environment to create only slower cars. The per-half-second new car creation probability was still 0.5. Since there were no faster cars, the cars were distributed uniformly across the four lanes. I also modified the sensory motor system to remove those actions and dimensions that are only relevant to looking backwards; these are: the gaze-backwards action, and the gaze-direction and gaze-speed perception dimensions.

I tested my hand-written policy in this environment. Over the course of 5000 steps, my policy made 99 collisions.

An agent that chooses random actions made 788 collisions.

Then I trained U-Tree in the same environment. U-Tree was trained for 10,000 steps, or 83 simulated minutes of experience. The utile distinction test used Kolmogorov-Smirnov probability of $p = 0.0001$, the number of steps taken between splitting tests was $k = 1000$. For the first 2000 steps, the exploration probability was 1.0; for the second 4000 steps, it was 0.4; then 2000 steps at 0.2; and 2000 steps at 0.1. This exploration schedule is fairly arbitrary. I found that it was important to explore a lot in the beginning, and explore less during later steps, but I did not perform a wide array of experiments to determine the best exploration schedule. The values for p and k are also fairly arbitrary.

During a 5,000 step test run after learning, the policy learned by U-Tree made 67 collisions. This represents a 32 percent improvement of my hand-coded policy, and a 91 percent improvement over the random policy. The learned policy has 51 leaves. Features that are not relevant to the task, like color, for example, are never used to distinguish states.

It would have been especially satisfying if U-Tree learned a tree that both performed better and was smaller than my hand-written tree.

The first newly created leaf was

```
t=0   perception dimension [Gaze object = Car]
t=0   perception dimension [Gaze side = Center]
t=0   perception dimension [Gaze distance = Nose]
```

This format for labeling leaves will be used throughout the rest of this section and the appendices that show experimental results. The first column, `t=0`, indicates that the history index is zero for that distinction, (*i.e.* this branch makes a distinction based on the current percept). If the distinction were looking one step backward in time, the column would indicate `t=1`. This distinction was added because it predicted an imminent collision. The brackets contain the label for the perceptual dimension and the value of that dimension.

The predictive capability was refined by the second leaf created:

```
t=0   per [Gaze object = Car]
t=0   per [Gaze side = Center]
t=0   per [Gaze distance = Nose]
t=0   per [Gaze refined dist = Far-half]
```

A few distinctions later it added

```
t=0   per [Gaze object = Car]
t=0   per [Gaze side = Left]
t=0   per [Gaze distance = Far]
```

based on the shift-gaze-lane action. It found that shifting left when the left-hand car was far had higher reward than shifting when the car was near.

The agent performs better than the policy I wrote. It also seems to drive much more aggressively. Sometimes it lets cars get close, and swerves out of their way at the last half-second or two. I think the agent has actually learned to take advantage of the fact that, because of the way new cars are randomly added to the environment, no two cars will ever appear at the same distance forward from the agent. Thus, if the agent comes right up to the tail of one car and then shifts lanes at the last half-second to arrive side-by-side with that car, it can be guaranteed that there will be no other car there.

Here is an action sequence that demonstrates swerving in and out of traffic.

The agent is looking ahead and sees that at an intermediate distance, there is a car. The agent chooses just to look at it again.

```
t=0   per [Gaze object = Car]
t=0   per [Gaze side = Center]
t=0   per [Gaze distance = Near]
t=0   per [Gaze refined dist = Far-half]
                             Choosing action:  Gaze Center
```

The agent sees that the car is now in the close-half of the “near” distance measure, and chooses to look right in preparation for getting out of the way.

```
t=0   per [Gaze object = Car]
t=0   per [Gaze side = Center]
t=0   per [Gaze distance = Near]
t=0   per [Gaze refined dist = Close-half]
                        Choosing action:  Gaze right
```

The agent sees the shoulder on the right, it cannot shift in this direction. It remembers that it just saw a car straight ahead, and it was near—there is no time to waste, the car ahead is now at distance "nose, far-half." The agent looks left.

```
t=0   per [Gaze object = Shoulder]
t=0   per [Gaze side = Right]
t=1   per [Gaze object = Car]
t=1   per [Gaze side = Center]
t=1   per [Gaze distance = Near]
                        Choosing action:  Gaze left
```

The agent sees a car on the left, but it is far away, so it can shift to this lane and still have time to get out of its way later. (Meanwhile, there is no more time to avoid the car ahead; the agent is right on its tail at distance "nose, close-half.") The agent chooses to shift into the lane it is looking at.

```
t=0   per [Gaze object = Car]
t=0   per [Gaze side = Left]
t=0   per [Gaze distance = Far]
t=0   per [Gaze refined dist = Far-half]
                        Choosing action:  Goto gaze-lane
```

The agent just changed lanes, and is now looking straight ahead in its new lane. It sees the car it saw before, and now it's one step closer. The agent chooses to look right in preparation for getting out of the way.

```
t=0   per [Gaze object = Car]
t=0   per [Gaze side = Center]
t=0   per [Gaze distance = Far]
t=0   per [Gaze refined dist = Close-half]
                        Choosing action:  Gaze right
```

The agent sees that the right lane is clear. The agent chooses to shift into the lane that it is looking at.

```
t=0   per [Gaze object = Road]
t=0   per [Gaze side = Right]
                        Choosing action:  Goto gaze-lane
```

The agent just changed lanes, and is now looking straight ahead in its new lane. The lane is clear. The agent chooses to continue looking straight ahead—on the watch for future obstacle cars in its lane.

```
t=0   per [Gaze object = Road]
t=0   per [Gaze side = Center]
                        Choosing action:  Gaze center
```

Learning to Drive Around Slower and Faster Cars

Although I have no human-written policy to compare against, I re-added the faster cars, and re-added the perception dimensions I had removed, then trained U-Tree on this task.

In this environment, over the course of 5000 steps, an agent that chooses random actions made 1260 collisions and spent 775 half-seconds getting honked at.

I trained U-Tree for 18,000 steps, or two and a half simulated hours of experience. The utile distinction test used Kolmogorov-Smirnov probability of $p = 0.0001$, the number of steps taken between splitting tests was $k = 1000$. For the first 2000 steps, the exploration probability was 1.0; for the second 2000 steps, it was 0.8; then 8000 steps at 0.4, 2000 steps at 0.2; and 2000 steps at 0.1. Again, this exploration schedule is fairly arbitrary—I did not perform a wide array of experiments to determine the best exploration schedule. The values for p and k are also fairly arbitrary.

During a 5000-step test trial, the policy learned by U-Tree made 280 collisions and got honked at for 176 half-seconds. The learned policy has 143 leaves.[1] The represents a 77 percent improvement in collision avoidance and a 77 percent improvement in honking avoidance over the random policy.

The first created leaf was

```
t=0   per [Hear horn = Yes]
```

The second created leaf was

```
t=0   per [Hear horn = No]
t=0   per [Gaze object = Car]
t=0   per [Gaze distance = Nose]
t=0   per [Gaze refined dist = Close-half]
```

Several splits later it added the following leaf. It was added because the agent found that, when it choose the goto-gaze-lane action from this leaf, the agent could more accurately predict high utility than when it did not distinguish based on gaze-size and gaze-object.

```
t=0   per [Hear horn = Yes]
t=0   per [Gaze side = Right]
t=0   per [Gaze object = Road]
```

[1]More leaves would have been created if I let the training run longer. I chose this stopping point rather arbitrarily. Growth of the tree could have been more efficient given better exploration and better search of the fringe. Sections 8.2.7 and 8.2.9 discuss these two issues.

A print out of the full policy tree containing 143 leaves is too large to be included in this document. The tree and other experimental data can be obtained by contacting the author.

Here is an action sequence in which the agent demonstrates its ability move out of the way of faster cars before they start honking. The agent does this by alternately looking forward and backward whenever it sees no cars. When it sees a slower car in front, or a faster car behind, the agent gets out of the way by shifting lanes.

This sequence is particularly interesting because it demonstrates how the agent shifts to avoid looming cars, but stays in its lane when it sees receding cars.

In this first step, the agent is looking forward in its lane and sees the road on the horizon—the coast is clear. The agent chooses to look backwards, continuing the cycle of forward/backward gaze shifts that has been going on for the past several time steps. (Note that the agent remembers looking backwards and seeing the road one time step ago, so it knows that there is no impending honking.)

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Looming]
t=0   per [Gaze distance = Far]
t=0   per [Gaze refined dist = Far-half]
t=0   per [Gaze object = Road]
t=0   act [Goto gaze-lane]
t=1   per [Hear horn = No]
t=1   per [Gaze object = Road]
t=1   per [Gaze direction = Backward]
                            Choosing action:  Gaze backward
```

Looking backwards, the agent sees the road. The coast is clear behind also. The agent continues the cycle by looking forward center. The agent also knows that there is not an urgent situation because it remembers having seen the road ahead one time step ago.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Receding]
t=1   per [Gaze side = Center]
t=1   per [Gaze direction = Forward]
t=1   per [Gaze distance = Far]
t=0   per [Gaze direction = Backward]
t=1   per [Gaze object = Road]
t=0   per [Gaze object = Road]
t=1   act [Goto gaze-lane]
                            Choosing action:  Gaze forward center
```

Again, the coast is clear ahead. (Note that, although the agent alternates between looking forward and backward whenever there are no cars in its lane, the cycle consists of more than two agent internal states. There is no particular reason for this—it is simply the policy that the agent's limited exploration happens to have stumbled upon.) The agent continues the cycle by looking backwards.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Looming]
t=0   per [Gaze distance = Far]
t=0   per [Gaze refined dist = Far-half]
t=0   per [Gaze object = Road]
t=0   act [Gaze forward center]
t=1   per [Gaze speed = Receding]
t=1   per [Hear horn = No]
t=1   per [Gaze object = Road]
t=2   act [Goto gaze-lane]
                        Choosing action:  Gaze backward
```

A car has appeared on the horizon behind the agent. The car is far away, though, and, as in the slow-car-only trace, the agent simply looks at it again and lets it get one step closer.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Looming]
t=0   per [Gaze distance = Far]
t=0   per [Gaze refined dist = Far-half]
t=0   per [Gaze object = Car]
t=1   per [Gaze object = Road]
t=0   per [Gaze direction = Backward]
                        Choosing action:  Gaze backward
```

Now the agent has seen the car behind for two time steps. It chooses to look right and preparation for getting out of the way.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Looming]
t=0   per [Gaze distance = Far]
t=0   per [Gaze refined dist = Far-half]
t=0   per [Gaze object = Car]
t=1   per [Gaze object = Car]
t=0   per [Gaze direction = Backward]
                        Choosing action:  Gaze forward right
```

Looking forward in the right lane, the agent sees no cars, and chooses to shift right. Also, the agent remembers having seen a car in the last time step, so it knows that the situation may be urgent.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Right]
t=0   per [Gaze object = Road]
t=0   per [Gaze direction = Forward]
t=1   per [Gaze object = Car]
                         Choosing action:  Goto gaze lane
```

From the new lane, the agent re-starts its forward/backward gaze cycle. The agent looks backward. The agent's memory tells it that it has recently shifted into this lane. This memory distinguishes the current situation from one in which it knows that there is no looming car on the agent's tail—a situation in which it could get honked at.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Looming]
t=0   per [Gaze distance = Far]
t=0   per [Gaze refined dist = Far-half]
t=0   per [Gaze object = Road]
t=0   act [Goto gaze-lane]
t=1   per [Hear horn = No]
t=1   per [Gaze object = Road]
t=1   per [Gaze direction = Forward]
t=1   per [Gaze side = Right]
                         Choosing action:  Gaze backward
```

The agent sees another car behind it, but the agent notices that the car is receding, not looming, and thus it does not begin its "get out of the way" behavior. It chooses goto-gaze-lane, which, when looking in the center lane, has the identical effect as gaze-forward-center. The agent is merely continuing it forward/backward gaze cycle.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Receding]
t=1   per [Gaze side = Center]
t=1   per [Gaze direction = Forward]
t=1   per [Gaze distance = Far]
t=0   per [Gaze direction = Backward]
t=1   per [Gaze object = Road]
t=0   per [Gaze object = Car]
t=0   per [Gaze distance = Near]
t=0   per [Gaze refined dist = Close-half]
                          Choosing action:  Goto gaze lane
```

The agent sees no cars in front either, and so continues the cycle.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Looming]
t=0   per [Gaze distance = Far]
t=0   per [Gaze refined dist = Far-half]
t=0   per [Gaze object = Road]
t=0   act [Goto gaze-lane]
t=1   per [Hear horn = No]
t=1   per [Gaze object = Car]
t=1   per [Gaze speed = Receding]
t=1   per [Gaze distance = Near]
t=1   per [Gaze refined dist = Close-half]
                        Choosing action:  Gaze backward
```

In this next sequence the agent demonstrates its ability to avoid slower cars. The sequence is similar to the sequence shown on page 101, in which the agent was trained with a simplified sensory system and an environment with no faster cars. Here, however, we see that the agent must distinguish between looming and receding cars.

In this first step, the agent sees a car in front of it. The car is near and looming. The agent looks left, in preparation for swerving to avoid the car.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Looming]
t=0   per [Gaze distance = Near]
t=0   per [Gaze direction = Forward]
t=0   per [Gaze refined dist = Close-half]
                        Choosing action:  Gaze Left
```

The agent sees a car to the left, but it is far, so it shifts into that lane anyway.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Left]
t=0   per [Gaze object = Car]
t=0   per [Gaze direction = Forward]
t=0   per [Gaze distance = Far]
t=0   per [Gaze speed = Looming]
t=0   per [Gaze refined dist = Far-half]
                        Choosing action:  Goto Gaze Lane
```

The agent is looking forward center and sees the same car it saw before, still far in the distance. The agent chooses goto-gaze-lane, which has the same effect as gaze-forward-center, since the agent is looking in its own lane.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Looming]
t=0   per [Gaze distance = Far]
t=0   per [Gaze refined dist = Far-half]
t=0   per [Gaze object = Car]
t=1   per [Gaze object = Car]
t=0   per [Gaze direction = Forward]
t=1   per [Hear horn = No]
t=1   per [Gaze side = Left]
                     Choosing action:  Goto Gaze Lane
```

For two steps more the agent lets the approaching car get closer.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Looming]
t=0   per [Gaze distance = Far]
t=0   per [Gaze refined dist = Close-half]
t=0   per [Gaze direction = Forward]
                     Choosing action:  Goto Gaze Lane

t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Looming]
t=0   per [Gaze distance = Near]
t=0   per [Gaze direction = Forward]
t=0   per [Gaze refined dist = Far-half]
                     Choosing action:  Goto Gaze Lane
```

The agent looks left in preparation for passing the slower car.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Looming]
t=0   per [Gaze distance = Near]
t=0   per [Gaze direction = Forward]
t=0   per [Gaze refined dist = Close-half]
                     Choosing action:  Gaze Left
```

The left lane is clear, so the agent shifts into that lane.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Left]
t=0   per [Gaze object = Road]
```

```
t=0   per [Gaze direction = Forward]
t=1   per [Hear horn = No]
t=1   per [Gaze direction = Forward]
t=2   per [Hear horn = No]
                        Choosing action:  Goto Gaze Lane
```

The new lane is still clear, so the agent begins is cycle of looking forward and backward. The agent gazes backward, then forward, the backward.

```
t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Looming]
t=0   per [Gaze distance = Far]
t=0   per [Gaze refined dist = Far-half]
t=0   per [Gaze object = Road]
t=0   act [Goto gaze-lane]
t=1   per [Hear horn = No]
t=1   per [Gaze object = Road]
t=1   per [Gaze direction = Forward]
t=1   per [Gaze side = Left]
                        Choosing action:  Gaze backward

t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Receding]
t=1   per [Gaze side = Center]
t=1   per [Gaze direction = Forward]
t=1   per [Gaze distance = Far]
t=0   per [Gaze direction = Backward]
t=1   per [Gaze object = Road]
t=0   per [Gaze object = Road]
t=1   act [Goto gaze-lane]
                        Choosing action:  Gaze forward center

t=0   per [Hear horn = No]
t=0   per [Gaze side = Center]
t=0   per [Gaze speed = Looming]
t=0   per [Gaze distance = Far]
t=0   per [Gaze refined dist = Far-half]
t=0   per [Gaze object = Road]
t=0   act [Gaze forward center]
t=1   per [Gaze speed = Receding]
t=1   per [Hear horn = No]
t=1   per [Gaze object = Road]
t=2   act [Goto gaze-lane]
                        Choosing action:  Gaze backward
```

Performance Discussion

The agent learns to successfully avoid slower cars and move out of the way of honking faster cars. The agent, however, does not always avoid scrapes with slower cars. Sometimes the randomly created configuration of slower cars results in a situation in which it is impossible to avoid a scrape; sometimes the agent's lack of good exploration results in a policy that doesn't handle certain avoidable patterns of obstacle cars. Improving exploration is discussed in section 7.6 and section 8.2.7.

Some of the conjunctions may seem odd and uninterpretable. Remember that there is some context contained in the agent's policy. We may look at a conjunction and think of necessary conditions that aren't tested for there, but the agent's policy may only bring the agent to this state in situations in which that condition holds.

Perhaps, more likely, however, is that the agent simply builds some fairly bizarre distinctions. If one examines the states and the statistics that caused the split, one can verify that all of them are utile, but splits can still be less than optimal because (1) exploration is poor, (2) no information-theoretic technique is used to make the splits in an optimal order, (3) the agent has circular dependency between utility, distinctions and the the policy. Section 8.2.7 discusses exploration. Section 8.2.9 discusses information-theoretic techniques. Section 7.6 discusses the circular dependency.

The fact that so much data is required only underlines the benefit of instance-based learning. If the agent had to start all over again, re-gathering this large amount of data every time a split was made, the amount of experience required would be ridiculously large.

Here is one amusing behavior the agent learns: whenever a crash is imminent and unavoidable, the agent learns to choose the gaze-backwards action! I equate this with a scared person covering their eyes. I believe that the reinforcement learning mechanism behind this action choice is that looking backwards puts the agent in a perceptual state that is not *always* associated with imminent punishment. With enough training and enough data, the agent will uncover the hidden state associated with "covering its eyes" in these situations. It will add a new branch that distinguishes between looking backwards after seeing an imminent crash and looking backwards after having come from other situations.

7.5 Related Work with Trees

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

7.5.1 Prediction Suffix Tree

USM has more in common with Prediction Suffix Tree Learning (PSTL) than any other algorithm. From it USM borrows directly the use of a tree to represent variable amounts of memory, and the use of a fringe to test for statistically significant predictions.

To this common base, USM adds several features that are specific to reinforcement learning. We add a notion of actions, where layers of the tree alternately add distinctions based on previous actions and observations. We add reward to the model, and set up the statistical test to add distinctions based on future discounted reward, not observations. Because reward values are continuous we must use a statistical test that can handle continuous values, not discrete values. We also add dynamic programming in order to implement value iteration.

Furthermore, note that, unlike PSTL, the training sequences are not given to USM ahead of time. USM must generate the training sequences using its current suffix tree. Thus, the learning algorithm is not only responsible for building the tree, it must also explore the environment in order to generate sequences relevant for learning the tree.

7.5.2 Parti-game

The key feature U-Tree inherits from Parti-game is its instance-based foundation. Both algorithms take advantage of memorized instances in order to speed learning. Unlike PSTL, both U-Tree and Parti-game must generate their instances on-line.

The similarities between U-Tree and Parti-game also provide an insight into the similarities between learning with hidden state and learning in continuous spaces. In each case the agent must learn the granularity of a state space—U-Tree divides its state space by increasingly detailed distinctions of history, Parti-game divides its state space by increasingly fine distinctions in multidimensional continuous space. The success of instance-based techniques on problems in continuous spaces [Moore, 1992; Schaal and Atkeson, 1994; Schneider, 1994] give further support to the statements in chapter 5 regarding the reasons that instance-based methods are well-suited to problems in which the state space granularity is changing.

Parti-game differs from U-Tree in several ways. In Parti-game the agent has available a greedy controller that moves the agent towards the global goal state; new distinctions are added when the greedy controller fails. In U-Tree, the agent has no greedy controller and works with the traditional local rewards from the environment; new distinctions are added when statistics indicate that doing so will help predict return. U-Tree handles noisy rewards and actions well; Parti-game requires deterministic environments.

The most obvious difference is that Parti-game's tree represents distinctions in a continuous geometric space and U-Tree's tree represents distinctions in short-term memory and discrete perceptual space.

7.5.3 G-algorithm

From the G-algorithm, U-Tree inherits its use of a robust statistical test on reward values. Both algorithms apply robust statistical tests to reward values, and thus, unlike Parti-game, both U-Tree and the G-algorithm can handle noise.

Unlike U-Tree and Parti-game, the G-algorithm is not instance-based. This results in a significant inefficiency whenever a distinction is added: each time the G-algorithm splits a state, the G-algorithm is forced to reset the child state's Q-values to zero; (see [Chapman and Kaelbling, 1991], end of section 3.1). Because the G-algorithm has no raw record of the previous experience, it cannot obtain more detail about the old data for its reorganization into the new distinctions, and thus it cannot know how to distribute the agglomerated experience from the parent state into the new children states. Each time a state is split, the G-algorithm throws away all accumulated experience for that region, and must effectively re-learn that region of state space from scratch.

Another difference between the G-algorithm and U-Tree parallels a difference between UDM and U-Tree: both the G-algorithm and UDM test the the benefit of only a single split at a time. Due to this limitation, the G-algorithm and UDM require that the relevance of significant distinctions be detectable in isolation. If a series of distinctions are relevant for the task, but the distinctions in isolation are not useful in and of themselves, the G-algorithm and UDM will fail; (see [Chapman and Kaelbling, 1991], end of section 2; [McCallum, 1993b], section 5). U-Tree, on the other hand, uses a multi-step fringe and thus can test for the relevance of several distinctions (in conjunction) at the same time.

7.5.4 Variable Resolution Dynamic Programming

Unlike PSTL, Parti-game or the G-algorithm, Variable Resolution Dynamic Programming (VRDP) and U-Tree use dynamic programming directly on a learned model of the relevant parts of the environment.

There are advantages to separating the model learning, which requires exploration, from the "non-learning" dynamic programming, which only requires computation. For instance, with such a scheme, reward can be easily propagated from crucial regions of state space that are, nevertheless, difficult to visit multiple times. In Q-learning–based algorithms, reward propagation requires multiple runs through the propagation path. Even in path play-back schemes like Experience Replay [Lin, 1991a], the agent must still visit the reward state multiple times in order to propagate reward out along the multiple different paths that may be available. In addition to U-Tree and VRDP, several other reinforcement learning algorithms take advantage of a model-based approach [Sutton, 1990a; Barto *et al.*, 1991; Moore and Atkeson, 1993].

A disadvantage of VRDP is that is it partitions the state space into high resolution *everywhere* the agent visits, potentially making many irrelevant distinctions. U-Tree, on the other hand, creates only utile distinctions.

7.6 Discussion

U-Tree successfully combines instance-based learning, statistical techniques robust to noise, dynamic programming on a learned model of the environment, variable length short-memory for overcoming hidden state, feature-selection for implementing selective perception, and utile distinctions. The algorithm seems to fit many complex puzzle pieces together smoothly, and the preliminary experimental results look promising.

The limitations of U-Tree and ideas for its improvement are discussed in detail in section 8.2. In this section I briefly introduce a few of the relevant issues.

Exploration

The current implementation of U-Tree does not explore its environment efficiently. The agent explores by choosing a randomly selected action with probability e at each time step. Schemes based on statistics [Kaelbling, 1990a] or on counters [Thrun, 1992b] would perform much better. However, these schemes cannot be straightforwardly applied to tasks with hidden state because state aliasing can result in the agent thinking that it has much experience with a state that is, in fact, has not been previously visited by the agent.

Section 8.2.7 discusses a specific idea for improved exploration with hidden state.

The Curse of Dimensionality

There is no escape from the curse of dimensionality.

By adding distinctions based on only one dimension at a time, U-Tree has the capability to hide the curse of dimensionality from from the agent's internal state space, and thus from the dynamic programming calculations. This, however, simply means that the agent must deal with the curse at another level. When the agent attempts to expand the fringe and test possible hypothesis distinctions, the agent faces the full brunt of the curse. The tricks described in on page 93 can help prune some of the large number of possible expansions, but still leave the agent with a significant computational load.

Section 8.2.9 discusses the use of an information theoretic approach that could make the process of searching the fringe more efficient. Moore and Lee discuss efficient ways to use cross validation to search for appropriate models in [Moore and Lee, 1994]. Section 8.2.10 discusses an idea for avoiding the use of a fringe entirely.

Non-optimal trees

U-Tree does not learn optimal trees in any reasonable sense of the term "optimal." We could, for example, define an ϵ-optimal tree to be the tree with minimal number of nodes that can represent a policy with a utility that is within ϵ of the policy optimal.

There are two reasons it does not learn optimal trees:

- U-Tree adds distinctions in a greedy way. Instead of choosing the order of the nested distinctions optimally, it simply searches the possible distinctions in some pre-defined order, and instantiates the first distinction it finds that passes the test. Section 8.2.9 discusses ways to improve the situation by the application of information-theoretic approaches.

- U-Tree suffers from a "chicken and egg problem" with respect to utility, the policy and state distinctions. Which distinctions the agent makes depends on the utility statistics; the utility depends on the policy; the policy depends on the state distinctions. The agent approaches this three-step circular dependency by desperately trying to pull itself up by its own bootstraps, and hoping for the best. Improved exploration strategies should help; see section 8.2.7.

8 Conclusions

Chapter Outline

This chapter describes the situations in which this dissertation's approach is likely to succeed or fail, provides detailed ideas for further enhancement, and discusses the impact of this work.

8.1 Applying this Work

In what situations can we expect the approach used by U-Tree to succeed? In what situations will it fail? One advantage of U-Tree is that, because the technique is based on perspicuous trees and statistics, we can say something in response to these questions.

Here are situations in which U-Tree's performance is limited. In some cases, I mention techniques that could perform better.

- **Difficulty with long memories.**

  U-Tree will fail to uncover utile hidden state if that utile hidden state is correlated with percepts or actions that occur more time steps back than the agent searches in the fringe—that is, if the task is k-step-Markov, but not $(k-1)$-step-Markov, if the agent only expands the fringe to a history window of size less than k, and if there is no utile hidden state less than k steps back.

  Some algorithm that looked back a large, variable number of steps, or that clustered data into groups by utility and then searched for perceptual, action and history differences that formed correct boundaries, might work better. Section 8.2.8 discusses this second idea.

  Note that some utile hidden state may be invisible hidden state—and in that case no algorithm can uncover it.

- **Difficulty with large conjunctions.**

U-Tree will fail to find utile conjunctions of features if the conjunction requires more terms than the number of nested layers in the agent's expansion of the fringe—that is, if the utile conjunction includes more than k terms, and if the agent only expands its fringe to a depth less than k, and if no sub-conjunction with less than k-terms is a utile distinction.

This is a reiteration of the problem of detectable relevance of pieces of information in isolation, discussed in section 7.5.3.

- **Difficulty with rewards that are difficult to find.**

  U-Tree will fail to make utile distinctions if it cannot gather enough data to measure sufficient statistical significance. The agent may not be able to gather enough data with different rewards if the task involves a goal of achievement that is extremely difficult to fall upon by a random walk, if there is no intermediate reward that leads the agent to the goal, and if the agent has no teacher.

  A bottom-up approach based on merging, as opposed to splitting, might work better in these situations. Section 8.2.6 discusses the use of a bottom-up technique.

- **Difficulty with loops in the environment.**

  U-Tree will fail to represent utile memory distinctions efficiently if the task requires the agent to remember a feature across a loop in the environment. While a finite state machine can directly represent the concept of a state transition loop, linear history windows, (including those with variable length windows), cannot represent memory across loops efficiently. U-Tree, like all suffix-tree-based techniques, must build separate tree branches for each possible number of times through the loop.

Section 8.2 explains more weaknesses of the U-Tree method, and discusses ideas for improvement.

Here are some situations in which the U-Tree approach will work well. I particularly emphasize situations in which some other techniques do not perform well.

- **Success with large perception spaces.**

  U-Tree can perform well even when the number of percepts is large. Some other techniques perform well with a small number of percepts, but buckle under the burden of large state spaces because they use an agent-internal state space that is as large as the perceptual state space. U-Tree can handle larger perception spaces because the utile distinction tree hides the full size of the perceptual space from the agent's policy. (U-Tree is not, however, impervious to the size of the perceptual state space. As always, there is no escape from the curse of dimensionality. The amount of computation necessary for testing expansions of the fringe depends on the number of perceptual dimensions and the size of those dimensions. Relatively, however, U-Tree is not as sensitive to perceptual space size as many other reinforcement learning algorithms—especially those that deal with hidden state.)

Other techniques used to represent reinforcement learning state spaces, such as neural networks and CMAC's [Albus, 1975] do not suffer from a one-to-one correspondence between percepts and agent states, but occasionally have problems because they cannot represent as small a granularity as is necessary—their representational power is limited by the number of units or hidden layers in the network, or the size of the CMAC's window. U-Tree, on the other hand, can represent variable granularity—right down to the resolution of the sensors—without any interference.

- **Success with hidden state.**

  U-Tree can perform well even when the task involves utile hidden state. Unlike many other reinforcement learning algorithms, non-Markovian utile hidden state need not be a barrier to good performance. U-Tree will add memory to its state space in order to create a Markovian agent internal state space from a non-Markovian perceptual space. This applies to hidden state created by both active and non-active perceptual systems.

- **Success with noise.**

  U-Tree can perform well even when there is noise in the environment. The algorithm can handle noisy perception, action and reward—and can distinguish between variations caused by noise and those caused by structure. It does this by using statistical tests that are robust to noise.

- **Success with expensive experience.**

  U-Tree will work efficiently when experience is more expensive than computation. It does this by using instance-based learning, which keeps all data from previous experience.

- **Applicability to general reinforcement learning domains.**

  Unlike some other algorithms that are limited to restricted reinforcement learning domains, U-Tree can handle state transitions and delayed reward; it can handle both goals of achievement and goals of maintenance; it can handle domains where there is no distinction between perceptual and manipulative actions.

8.2 Future Extensions

There are several interesting possibilities for further development of algorithms that build on U-Tree as a base.

8.2.1 Sophisticated and Biased Utile Distinctions

History distinctions do not have to be purely linear strings of experience. They can make distinctions based on more sophisticated questions like "Did X happen in the last N steps?", or "Did X happen more recently than Y?" The same principle applies to perceptual distinctions. The fringe can contain distinctions based on questions like "Is the color in the fovea more X than Y?", or "Is the tension on joint X greater than the tension on joint Y?"

If the agent designer has domain knowledge about which distinctions might be useful, U-Tree provides a straightforward mechanism for taking advantage of it: the designer can add biased distinctions to the list of distinctions the fringe will test.

The designer does not have to know with certainty which distinctions will be useful—the statistical tests will ensure that the useless distinctions don't force the agent to re-learn its policy in multiple uselessly distinguished states. However, the designer can potentially help the agent make complex, sophisticated, useful distinctions by peppering the set of distinctions with educated guesses.

8.2.2 Utile Distinctions in Continuous Space

The current implementation of U-Tree uses a tree to represent distinctions in a discrete sensory space, but the same technique could be used to represent distinctions in continuous space. The tree branches would correspond to nested "threshold" boundaries in continuous space. Parti-game [Moore, 1993] is an example of an algorithm that uses a tree to make distinctions in a continuous space.

The fringe could consist either of partitions at midpoints, as in Moore's Parti-game [Moore, 1993], or of partitions at points determined by cluster boundaries, as in Fayyad's work [Fayyad *et al.*, 1993].

8.2.3 Utile Distinctions in Action Space

Currently, U-Tree selects from among possible perceptual distinctions, but there is no reason why it could not also select from among possible action distinctions.

If a large action space were divided into separate dimensions, (with, for example, different dimensions sending signals to a robot's wheels, arm, hand, and speaker), U-Tree could select from among different action dimensions in precisely the same way it does from different perception dimensions. Like the perceptual distinction scheme, this would allow the agent to reduce the branching factor on nodes that make state distinctions based on past actions.

I have just described how the agent could make utile state distinctions based on past actions, but we may also want to make utile distinctions in the space of future actions. I imagine a new algorithm in which the leaves of the tree would not contain a Q-table with one entry per action, but instead would contain another tree of nested action distinctions, and each leaf of this tree would contain a single Q-value.

8.2.4 Explicitly representing noisy observation

Probabilistic Suffix Trees cannot not handle noisy observations as well as finite-state-automata, belief-vector-using approaches. That is not to say that U-Tree does not handle noisy observations at all; PST's may simply suffer in comparison because they do not explicitly represent functions that map state-observation pairs to probabilities. U-Tree handles noisy observations as well as standard Q-learning does, and in the same way as standard Q-learning does—by calculating weighted averages of return based on the number of times a state produced a certain observation.

One can make arguments against using full HMM's and POMDP's. They are more complex, and they are much more difficult to learn than PST's [Ron *et al.*, 1994; Abe and Warmuth, 1992; Gillman and Sipser, 1991; Littman, 1994a]. There are also arguments that modeling noisy observation is less important than modeling noisy actions and rewards [Dean *et al.*, 1992]. There has also been some success with methods that produce reliable observation outputs from very noisy sensors by handling the noise at a lower level [Mataric, 1990].

Still, if other considerations require explicit representation of stochastic observations, there may be ways of incorporating explicit representation of noisy observation into Probabilistic Suffix Trees without incurring the full cost of HMM's or POMDP's. If the nodes of the tree contained observation probability functions and the agent followed all branches of the tree instead of just one, the resulting mechanism would be like a Finite Suffix POMDP—a POMDP that did not represent memories that crossed loops; a suffix tree that represented explicit knowledge about observation noise. The idea is closely related to a probabilistic decision tree, in which, instead of "falling down" only one branch at a time, the agent "falls down" many branches, each with a different probability. (For example, see [Jordan, 1994].)

I do not consider the explicit representation of loops as critical because I believe repeated loops are best captured at the level of primitive actions. In fact, much previous work with reinforcement learning has already chosen to model loops as internal to the primitive actions [Mataric, 1994; Mahadevan and Connell, 1990; Moore and Atkeson, 1993]. Although I'm sure the ability to learn loop structures would be useful, in my current thinking, I find that the difficulty of learning loops outweighs the benefits.

8.2.5 Better Statistical Tests

Is the Kolmogorov-Smirnov test the statistical test we want? It asks whether two distributions came from the same source. Perhaps, instead, we only care whether the two distributions have the same mean? After all, the agent's policy is based on the mean.

The Student-T test is a statistical technique that asks whether two distributions have the same mean. I tried using the Student-T test, but I had only limited success. The test assumes that the data comes from a Gaussian distribution. Utility data doesn't, and this seemed to cause the test to split when it should not have.

Note that in some situations we might care to distinguish different distributions with the same mean after all—for example, in situations in which we want the agent to be risk-averse [Heger, 1994].

If testing for same-source distributions is what we want, perhaps we could do a better job with alternatives to the Kolmogorov-Smirnov test [Bosq, 1996; Wolpert, 1995].

8.2.6 Bottom-up Model Building, Instead of Top-Down Modeling

Instead of beginning by making no distinctions and adding distinctions as statistical evidence is gathered, we could begin by treating all experience as distinct, and then merge experience as statistical evidence is gathered. I think of this alternative as "bottom-up" model building, as opposed to the current "top-down" model building. Whereas U-Tree assumes that everything is the same, unless statistics show otherwise, a bottom-up method would assume that everything is different, unless statistics show otherwise.

Bayesian Model Merging [Stolcke and Omohundro, 1992] is an example of the application of bottom-up model building to hidden Markov models.

A bottom-up approach would particularly help when there is a goal of achievement that is difficult to attain. U-Tree requires several, possibly many, experiences before it will have enough data to show enough statistical significance for a state split. Before it makes the state split the agent cannot solve the task—even when given an identical problem situation. A bottom-up technique will also require multiple experiences before it can generalize well, but at least the agent will be able to repeat the solution when faced with identical initial conditions and a lack of noise.

Thus, a bottom up technique could perform better than top-down techniques when the initial random walk leads to the goal only rarely, and when identical situations show up during training and performance. Even when the conditions are not identical to a previously found solution, following the previous (slightly inapplicable) action sequence may explore more fruitful regions of state space better than the top-down technique's random walk would.

Bottom-up model building could be particularly useful if the agent is attempting "learning by watching" [Whitehead, 1992; Kuniyoshi *et al.*, 1990; Ikeuchi and Suehiro, 1992]. In learning by watching, the teacher presents the agent with an action sequence that accomplishes the task. The teacher may provide several presentations under different conditions. Then, the agent is expected to generalize from the experience gained by watching the teacher. Since the number of teacher presentations is likely to be limited, bottom-up model merging could apply well to learning by watching for the same reasons it applied well to situations in which it is hard to stumble on reward randomly.

8.2.7 Better Exploration

One of the most important outstanding issues in hidden state that is not addressed by this dissertation is how the agent can explore efficiently in the face of hidden state. I give the name *hidden exploration* to the collection of difficulties that arise when standard exploration techniques are applied to tasks with hidden state.

Hidden state makes exploration much more difficult than it is without state aliasing because the agent cannot know whether it has previously visited a particular state or not. A state may look the same as one the agent has seen a thousand times, yet, in fact, be a different state, and only look the same due to a many-to-one mapping between world states and percepts. A state may look different from anything the agent has seen before, yet, in fact, be the same as a state the agent has seen a thousand times, due to a one-to-many mapping between world states and percepts in an active perceptual system.

The proposed idea for improving exploration with hidden state is to use whichever method you would use without hidden states—counting methods, confidence intervals, etcetera—however, keep those counts, and statistics associated with the states, in the hypothesis distinction state space, *i.e.* in the nodes of the fringe. In short, the best the agent can hope for is to fully explore its distinction hypothesis space; this idea would implement exactly that. I call the method *Hypothesis Space Exploration.* Results with this technique show improvement by a factor of two [McCallum, 1996].

Some research has been done on stochastic policies for operating in hidden state environments [Jaakkola *et al.*, 1995; Littman, 1994a]; and, while I consider this work to be related to exploration, I consider it an approach to exploitation more than exploration.

8.2.8 Reducing Tree Size with Utile-Clustered Branches

U-Tree greatly reduces tree fan-out in comparison to Utile Suffix Memory. However, perhaps the agent could reduce tree fan-out even further by clustering together branches that have the same utility. In other words, instead of distinguishing between all values of a particular dimension, the agent could cluster together those values that have the same utility.

The use of clustering may require some cleverness to assure that all prefixes of a distinction are in the tree.

8.2.9 Information-Theoretic Splitting

Currently, U-Tree finds new state distinctions by a greedy method. It searches the space of possible distinctions according to some fairly arbitrary search scheme, and as soon as it finds a distinction that passes the statistical test, it makes that split. It is possible that, although the agent only makes splits that help predict utility, the agent builds a tree that is larger than necessary.

An information-theoretic approach would instead search for an optimal order for the nested distinctions, so as to reduce the number of tree nodes [Rissanen, 1987]. Quinlan and Rivest have done work in building decision trees according to *minimum description length* principles [Quinland and Rivest, 1989; Quinlan, 1994; Quinlan, 1995].

8.2.10 Not Using a Fringe

Perhaps the agent could search the space of possible distinctions even more efficiently if we combined the information theoretic-techniques with an approach that did not use a fringe at all.

Instead of searching the space of all possible distinctions and then testing them for differences in reward, consider turning the distinction search technique on its head.

Instead of making hypothesis distinctions and then ask if the distinctions help predict reward, begin by clustering the data in reward space, and then searching for distinctions that separate the clusters.

I believe this may be closer to the way humans find "utile distinctions." We categorize situations by their outcome, and then look for features that that distinguish the categories.

8.3 Thesis Summary

This dissertation has explored the issues of selective perception and hidden state in the context of reinforcement learning. The dissertation advocated the study of these issues by explaining that agents are embedded in complex environments, but with limited perceptual resources. When the agent has "too much sensory data," it should budget its perceptual resources by selectively attending to those features that are relevant to the task at hand. When the agent's limited resources cause the agent to have "too little sensory data," it should use contextual information, in the form of short-term memory, to disambiguate enough state features that the task at hand can be solved.

The dissertation presented a new framework for thinking about how these seemingly opposite concepts are actually closely intertwined. On the one hand, selective perception can be understood as "purposefully creating hidden state." On the other hand, the way to overcome hidden state can be understood as selectively choosing which features to remember and which to forget.

Four algorithms have been described. Utile Distinction Memory draws on statistical techniques to implement a utile distinction test—introducing state distinctions only when doing so helps the agent predict reward. Nearest Sequence Memory uses instance-based learning to efficiently make use of experience when building a variable granularity model. Utile Suffix Memory extends a suffix tree algorithm such that it can be applied to reinforcement learning; it also combines the advantages of the previous two algorithms. U-Tree builds on Utile Suffix Memory, modifying it to make utile *perceptual* distinctions as well as utile *history* distinctions. U-Tree embodies the close relationship between selective perception and hidden state. In empirical results, the later algorithms have been shown to perform an order of magnitude better than previous algorithms. All the algorithms handle noise, delayed rewards and infinite horizon-tasks.

This dissertation (especially U-Tree) demonstrates that three fundamental problems in reinforcement learning—(1) choosing memories to uncover hidden state, (2) choosing features to build a factored state representation, and (3) value function approximation—are all, in a sense, fundamentally rooted in the same problem: finding utile state distinctions.

The thesis statements of this dissertation are:

- Hidden state and selective perception are intimately related, and can be explained in terms of each other. Agents benefit by dealing with them together.

- Agents in complex environments should make those distinctions that are necessary for performing the task; these are called "utile distinctions"—the distinctions that help predict utility. It is not recommended that the agent make all the distinctions necessary for predicting perception because this would increase the agent's storage and computational burden and unnecessarily reduce generalization.

  Agents should learn these distinctions, instead of having them hard coded, for all the same reasons we want agents to learn the task itself: we may not know ahead of time which distinctions are necessary; the relevant distinctions may change if the environment changes; programming all the details is tedious.

- Often experience is more expensive than computation. In these cases, the agent should use instance-based (or "memory-based") learning to learn a model of the environment, and perform dynamic programming on the model.

  The relatively smaller number of agent internal states created by a utile distinction approach may make model-learning and dynamic programming on the model feasible where it would not have previously been possible with typically large, fixed-granularity models or models built with perceptual distinctions.

Bibliography

[Abe and Warmuth, 1992] N. Abe and M. Warmuth. On the computational complexity of approximating distributions by probabilistic automata. *Machine Learning*, 9:205–260, 1992.

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

[Agre, 1985] Philip E. Agre. The structure of everyday life. Technical Report 267, MIT Working Paper, 1985.

[Albus, 1975] J. S. Albus. A new approach to manipulator control: cerebellar model articulation controller (cmac). *Transations of the ASME: Journal of Dynamic Systems, Measurement and Control*, September 1975.

[Albus, 1991] James S. Albus. Outline for a theory of intelligence. *IEEE Trans. on Systems, Man, and Cybernetics*, 21(3):473–509, 1991.

[Aloimonos *et al.*, 1988] J. Aloimonos, A. Bandopadhay, and I. Weiss. Active vision. *International Journal of Computer Vision*, 1 (4):333–356, 1988.

[Astrom, 1965] K. J. Astrom. Optimal control of Markov decision processes with incomplete state estimation. *J. Math. Anal. Appl.*, 10:174–205, 1965.

[Atkeson, 1992] Christopher G. Atkeson. Memory-based approaches to approximating continuous functions. In M. Casdagli and S. Eubank, editors, *Nonlinear Modeling and Forecasting*, pages 503–521. Addison Wesley, 1992.

[Bacchus *et al.*, 1996] Fahiem Bacchus, Craig Boutilier, and Adam Grove. Rewarding behaviors. In *Proceeding of the Fourteenth National Conference on Artificial Intelligence AAAI-96*. AAAI Press/The MIT Press, 1996.

[Bacchus *et al.*, 1997] Fahiem Bacchus, Craig Boutilier, and Adam Grove. Structured solution methods for non-Markovian decision processes. In *Proceeding of the Fifteenth National Conference on Artificial Intelligence AAAI-97*. AAAI Press/The MIT Press, 1997.

[Ballard and Brown, 1992] D. H. Ballard and C. M. Brown. Principles of animate vision. *Computer Vision, Graphics, and Image Processing: Image Understanding*, 56(1):3–21, 1992.

[Ballard *et al.*, 1996] D. H. Ballard, M. M. Hayhoe, P. K. Pook, and R. Rao. Deictic codes for the embodiment of cognition. *Brain and Behavioral Sciences*, 1996. [To appear – earlier version available as National Resource Laboratory for the study of Brain and Behavior TR95.1, January 1995, U. of Rochester].

[Ballard, 1989] Dana H. Ballard. Reference frames for animate vision. In *Eleventh International Joint Conference on Artificial Intelligence*, pages 1635–1641, August 1989.

[Barto *et al.*, 1983] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuron-like elements that can solve difficult learning control problems. *IEEE Trans. on Systems, Man, and Cybernetics*, 13(5):834–846, 1983.

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

[Barto *et al.*, 1995] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1):81–138, 1995.

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

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

[Bertsekas, 1976] Dimitri. P. Bertsekas. *Dyanmic Programming and Stochastic Control.* Academic Press, 1976.

[Booker *et al.*, 1989] L. Booker, D. E. Goldberg, and J. H. Holland. Classifier systems and genetic algorithms. *Artificial Intelligence*, 40(1-3):235–282, 1989.

[Bosq, 1996] Denis Bosq. *Nonparametric statistics for stochastic processes : estimation and prediction.* Springer, 1996.

[Boutilier and Dearden, 1997] Craig Boutilier and Richard Dearden. Approximating value trees in structured dynamic programming. In *Proceeding of the Fifteenth National Conference on Artificial Intelligence AAAI-97.* AAAI Press/The MIT Press, 1997.

[Boutilier and Poole, 1996] Craig Boutilier and David Poole. Computing optimal policies for partially observable decision processes using compact representations. In *Proceeding of the Fourteenth National Conference on Artificial Intelligence AAAI-96.* AAAI Press/The MIT Press, 1996.

[Boutilier *et al.*, 1995] Craig Boutilier, David Poole, and Moises Goldszmidt. Exploiting structure in policy construction. In *Proceeding of the Thirteenth National Conference on Artificial Intelligence AAAI-95*. AAAI Press/The MIT Press, 1995.

[Boutilier *et al.*, 1996] Craig Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. Context-specific independence in bayesian networks. In *Proceedings of the Twelfth International Conference on Uncertainty in Artificial Intelligence*, 1996.

[Boyan *et al.*, 1995] J. A. Boyan, A. W. Moore, and R. S. Sutton. Proceedings of the workshop on value function approximation, machine learning conference. Technical Report Technical Report CMU-CS-95-206, Carnegie Mellon, 1995. http://www.cs.cmu.edu/afs/cs/project/reinforcement/ml95/.

[Brooks, 1986] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, pages 14–22, April 1986.

[Brooks, 1987] Rodney A. Brooks. Intelligence without representation. In *Workshop on the Foundations of Artificial Intelligence*. Endicott House, Dedham Mass., 1987.

[Brooks, 1991] Rodney A. Brooks. Intelligence without reason. AI Memo No. 1293, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1991.

[Brown *et al.*, 1989] C.M. Brown, H. Durrant-Whyte, J. Leonard, and B. Rao. Centralized and decentralized kalman filter techniques for tracking, navigation, and control. Technical Report 277, University of Rochester Computer Science Dept., March 1989.

[Brown, 1992] C. Brown. Issues in selective perception. In *Proceedings: IEEE International Conference on Pattern Recognition*, 1992.

[Burt, 1988] Peter J. Burt. Smart sensing within a pyramid vision machine. *Proceedings of the IEEE*, 76(8):1006–1015, August 1988.

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

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

[Chapman, 1989] David Chapman. Penguins can make cake. *AI Magazine*, 10(4):45–50, 1989.

[Chrisman and Simmons, 1991] L. Chrisman and R. Simmons. Sensible planning: focusing perceptual attention. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, July 1991.

[Chrisman *et al.*, 1991] Lonnie Chrisman, Rich Caruana, and Wayne Carriker. Intelligent agent design issues: Internal agent state and incomplete perception. *Working Notes of the AAAI Fall Symposium: Sensory Aspects of Robotic Intelligence*, 1991.

[Chrisman, 1992a] Lonnie Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Tenth National Conference on Artificial Intelligence*, pages 183–188, 1992.

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

[Cliff and Ross, 1995] Dave Cliff and Susi Ross. Adding temporary memory to ZCS. *Adaptive Behavior*, 3(2):101–150, 1995.

[Clinton, 1993] Bill Clinton. Presidential inaugural address, Jaunuary 1993.

[Cohn *et al.*, 1996] D. A. Cohn, Z. Ghahramani, and M. I. Jordan. Active learning with statistical models. *Journal of Artificial Intelligence Research*, 4:129–145, 1996.

[Cohn, 1993] David Cohn. Queries and exploration using optimal experiment design. In *Advances of Neural Information Processing Systems (NIPS 6).* Morgan Kaufmann, 1993.

[Colombetti and Dorigo, 1994] M. Colombetti and M. Dorigo. Training agents to perform sequential behavior. *Adaptive Behavior*, 2(3):247–275, 1994.

[Coombs, 1992] David J. Coombs. *Real-Time Gaze Holding in Binocular Robot Vision.* PhD thesis, University of Rochester, 1992.

[Corman *et al.*, 1990] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. *Introdution to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990.

[Crites and Barto, 1995] R. Crites and A. Barto. Improving elevator performance using reinforcement learning. In *Advances of Neural Information Processing Systems (NIPS 8).* Morgan Kaufmann, 1995.

[Damazio, 1994] Antonio R. Damazio. *Descartes' Error: Emotion, Reason, and the Human Brain.* G. P. Putnam, 1994.

[Dean *et al.*, 1992] Thomas Dean, Dana Angluin, Kenneth Basye, Sean Engelson, Leslie Kaelbling, Evangelos Kokkevis, and Oded Maron. Inferring finite automata with stochastic output functions and an application to map learning. In *Tenth National Conference on Artificial Intelligence*, pages 208–214, 1992.

[Dempster *et al.*, 1977] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of Royal Statistical Society, Series B*, 39:1–38, 1977.

[Deng and Moore, 1995] K. Deng and A. W. Moore. Multiresolution instance-based learning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[Dorigo and Colombetti, 1994] M. Dorigo and M. Colombetti. Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence*, 71(2):321–370, 1994.

[Dorigo, 1992] M. Dorigo. Alecsys and the autonomouse: Learning to control a real robot by distributed classifier systems. Technical Report 92-011, Politecnico di Milano, Italy, 1992. (also, *Machine Learning*, in press).

[Fayyad *et al.*, 1993] Usama M. Fayyad, Nicholas Weir, and S. Djorgovski. Skicat: A machine learning system for automated cataloging of large scale sky surveys. In *The Proceedings of the Tenth International Machine Learning Conference*. Morgan Kaufmann Publishers, Inc., 1993.

[Forsyth *et al.*, 1991] David Forsyth, Joseph L. Mundy, Andrew Zisserman, and Charles Rothwell. Invariant descriptors for 3d object recognition and pose. In *Applications of Invariance in Computer Vision: Proceedings of the First DARPA-ESPRIT Workshop on Invariance*, March 1991.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[Garvey, 1976] Thomas D. Garvey. Perceptual strategies for purposive vision. Technical Note 117, SRI International, September 1976.

[Gillman and Sipser, 1991] D. Gillman and M. Sipser. Inference and minimization of hidden Markov chains. In *Proceedings of the Seventh Annual Workshop on Computational Learning Theory*, 1991.

[Heger, 1994] M. Heger. Consideration of risk in reinforcement learning. In *The Proceedings of the Eleventh International Machine Learning Conference*. Morgan Kaufmann Publishers, Inc., 1994.

[Hilgard and Marquis, 1961] Ernest Ropiequet Hilgard and Donald George Marquis. *Conditioning and Learning.* Appleton-Centurn-Crofts, second edition edition, 1961. Edited by Gregory A. Kimble.

[Holland, 1975] John H. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, Ann Arbor, MI, 1975.

[Howard, 1960] Ronald A. Howard. *Dynamic Programming and Markov Processes.* The M.I.T. Press, Massachusetts Institute of Technology, Cambridge, MA, 1960.

[Humphreys, 1996a] Mark Humphreys. Action selection methods using reinforcement learning. In *From Animals to Animats: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (SAB'96)*. MIT Press, 1996.

[Humphreys, 1996b] Mark Humphreys. *Action Selection methods using Reinforcement Learning*. PhD thesis, University of Cambridge, 1996.

[Ikeuchi and Suehiro, 1992] Katsushi Ikeuchi and Takashi Suehiro. Towards an assembly plan from observation. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, May 1992.

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

[Jordan and Rumelhart, 1990] Michael I. Jordan and David E. Rumelhart. Supervised learning with a distal teacher. Technical report, MIT, 1990.

[Jordan, 1994] Michael I. Jordan. A statistical approach to decision tree modeling. In M. Warmuth, editor, *Proceedings Computational Learning Theory*. ACM Press, 1994.

[Kaelbling *et al.*, 1995] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. An introduction to reinforcement learning. In Luc Steels, editor, *The Biology and Technology of Autonomous Systems*. Elsevier, 1995.

[Kaelbling, 1986] L. P. Kaelbling. An architecture for intelligent reactive systems. In *Reasoning About Actions and Plans: Proceedings of the 1986 Workshop*. Morgan Kaufmann, 1986.

[Kaelbling, 1990a] Leslie P. Kaelbling. *Learning in Embedded Systems*. PhD thesis, Stanford University, 1990.

[Kaelbling, 1990b] Leslie Pack Kaelbling. *Learning in Embedded Systems*. PhD thesis, Stanford University, 1990.

[Kaelbling, 1995] Leslie Kaelbling. (Personal Communication). Brown University, Department of Computer Science, September 1995.

[Karlsson, 1994] Jonas Karlsson. Task decomposition in reinforcement learning. In *Proceedings of the AAAI Spring Symposium on Goal-driven Learning*, 1994.

[Koza, 1992] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[Kuniyoshi *et al.*, 1990] Yasuo Kuniyoshi, Hirochika Inoue, and Masayuki Inaba. Design and implementation of a system that generates assembly programs from visual recognition of human action sequences. In *Proceedings of the IEEE International Workshop on Intelligent Robots and Systems*, pages 567–574, 1990.

[Levin, 1973] L. A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973.

[Lin and Mitchell, 1992] Long-Ji Lin and Tom M. Mitchell. Reinforcement learning with hidden states. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 271–280, 1992.

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

[Lin, 1991b] Long Ji Lin. Self-improvment based on reinforcement learning, planning, and teaching. In *Proceedings of the Eighth International Workshop on Machine Learning*, 1991.

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

[Littman *et al.*, 1995] Michael L. Littman, Anthony R. Cassandra, and Leslie Pack Kaelbling. Learning policies for partially observable environments: Scaling up. In *The Proceedings of the Twelfth International Machine Learning Conference.* Morgan Kaufmann Publishers, Inc., 1995.

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

[Littman, 1994b] Michael L. Littman. The witness algorithm: Solving partially observable Markov decision processes. Technical Report CS-94-40, Brown University, Department of Computer Science, 1994.

[Lovejoy, 1991] W. S. Lovejoy. A survey of algorithmic methods for partially observable markov decision processes. *Annals of Operations Research*, 28:47–66, 1991.

[Luenberger, 1979] David G. Luenberger. *Introduction to Dynamic Systems.* John Wiley and Sons, 1979.

[Maes and Brooks, 1990] Pattie Maes and Rodney A. Brooks. Learning to coordinate behaviors. In *Proceedings of AAAI-90*, pages 796–802, 1990.

[Mahadevan and Connell, 1990] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. Technical report, I.B.M. T.J. Watson Research Center, December 1990.

[Mahadevan and Connell, 1991] Sridhar Mahadevan and Jonathan Connell. Scaling reinforcement learning to robotics by exploiting the subsumption architecture. In *Proceedings of the Eighth International Workshop on Machine Learning*, 1991.

[Mahadevan, 1996] Sridhar Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning, Special Issue on Reinforcement Learning*, 22:159–196, 1996.

[Mataric, 1990] Maja J. Mataric. A distributed model for mobile robot environment-learning and navigation. Master's thesis, Massachusetts Institute of Technology, 1990. (Technical Report AI-TR 1228).

[Mataric, 1994] Maja Mataric. Reward functions for accelerated learning. In *The Proceedings of the Eleventh International Machine Learning Conference*. Morgan Kaufmann Publishers, Inc., 1994.

[McCallum, 1992] R. Andrew McCallum. Using transitional proximity for faster reinforcement learning. In *The Proceedings of the Ninth International Machine Learning Conference*. Morgan Kaufmann Publishers, Inc., 1992.

[McCallum, 1993a] R. Andrew McCallum. Learning with incomplete selective perception. Technical Report 453, University of Rochester Computer Science Dept., March 1993. PhD thesis proposal.

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

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

[McCallum, 1995a] R. Andrew McCallum. Instance-based state identification for reinforcement learning. In *Advances of Neural Information Processing Systems (NIPS 7)*, pages 377–384, 1995.

[McCallum, 1995b] R. Andrew McCallum. Instance-based utile distinctions for reinforcement learning. In *The Proceedings of the Twelfth International Machine Learning Conference*. Morgan Kaufmann Publishers, Inc., 1995.

[McCallum, 1996] Andrew Kachites McCallum. Efficient exploration in reinforcement learning with hidden state. Technical report, University of Rochester Computer Science Dept., 1996.

[Meeden *et al.*, 1993] Lisa Meeden, G. McGraw, and D. Blank. Emergent control and planning in an autonomous vehicle. In David S. Touretsky, editor, *Proceedings of the Fifteenth Annual Meeting of the Cognitive Science Society*, pages 735–740. Lawerence Erlbaum Associates, 1993.

[Minsky, 1954] Marvin L. Minsky. *Theory of neural-analog reinforcement systems and its application to the brain-model problem.* PhD thesis, Princeton University, 1954.

[Montague *et al.*, 1995] P. Read Montague, Peter Dayan, and Terrence J. Sejnowski. A framework for mesencephalic dopamine systems based on predictive hebbian learning. (submitted), August 1995.

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

[Moore and Lee, 1994] A. W. Moore and M. S. Lee. Efficient algorithms for minimizing cross validation error. In *The Proceedings of the Eleventh International Machine Learning Conference*. Morgan Kaufmann Publishers, Inc., 1994.

[Moore and Schneider, 1995] Andrew W. Moore and Jeff G. Schneider. Memory-based stochastic optimization. In *Advances of Neural Information Processing Systems (NIPS 8)*. Morgan Kaufmann, 1995.

[Moore *et al.*, 1995] Andrew W. Moore, C. G. Atkeson, and S. A. Schaal. Memory-based learning for control. *Artificial Intelligence Review*, 1995. (submitted).

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

[Moore, 1992] Andrew W. Moore. *Efficient Memory-based Learning for Robot Control*. PhD thesis, University of Cambridge, November 1992.

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

[Mozer and Bachrach, 1990] Michael Mozer and Jonathan Bachrach. Discovering the structure of a reactive environment by exploration. *Neural Computation*, 2:447–457, 1990.

[Pavlov, 1923] I. P. Pavlov. New researches on conditioned reflexes. *Science*, 58:359–361, 1923.

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

[Pomerleau, 1992] Dean A. Pomerleau. *Neural Network Perception for Mobile Robot Guidance*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1992.

[Pook and Ballard, 1992] Polly Pook and Dana H. Ballard. Sensing qualitative events to control manipulation. *Proceedings of the SPIE Sensor Fusion V Conference*, November 1992.

[Pook, 1995] Polly Pook. *Teleassistance: Using Deictic Gestures to Control Robot Action.* PhD thesis, Department of Computer Science, University of Rochester, 1995.

[Poritz, 1988] Alan B. Poritz. Hidden markov models: a guided tour. In *Proceedings of ICASSP '88*, April 1988.

[Press *et al.*, 1988] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1988.

[Pryor, 1984] Karen Pryor. *Don't Shoot the Dog!* Bantam Books, New York, NY, 1984.

[Quinlan, 1994] J. R. Quinlan. The minimum description length principle and categorical theories. In *The Proceedings of the Eleventh International Machine Learning Conference.* Morgan Kaufmann Publishers, Inc., 1994.

[Quinlan, 1995] J. R. Quinlan. MDL and categorical theories (continued). In *The Proceedings of the Twelfth International Machine Learning Conference.* Morgan Kaufmann Publishers, Inc., 1995.

[Quinland and Rivest, 1989] J. R. Quinland and R. L. Rivest. Inferring decision tress using the minimum description length principle. *Information and Computation*, 80:227–248, 1989.

[Rabiner, 1989] Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2), February 1989.

[Raibert and Craig, 1981] M. Raibert and J. Craig. Hybrid position/force control of manipulators. *ASME Journal of Dynamic Systems, Measurements, and Control*, June 1981.

[Reece and Shafer, 1992] D. A. Reece and S. A. Shafer. Planning for perception in robot driving. In *Proceedings of the DARPA Image Understanding Workshop*, pages 953–960, 1992.

[Reece, 1992] Douglas A. Reece. *Selective Perception for Robot Driving.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1992. TR CMU-CS-92-139.

[Rimey and Brown, 1991] Raymond D. Rimey and Christopher M. Brown. Controlling eye movements with hidden markov models. *International Journal of Computer Vision*, 7(1):47–66, November 1991.

[Rimey, 1993] R. D. Rimey. *Control of Selective Perception.* PhD thesis, Department of Computer Science, University of Rochester, 1993.

[Ring, 1994] Mark Ring. *Continual Learning in Reinforcement Environments.* PhD thesis, University of Texas at Austin, August 1994.

[Rissanen, 1987] J. Rissanen. Stochastic complexity. *Journal of Royal Statistical Society, Series B*, 49(3):223–239, 1987.

[Rivest and Schapire, 1987a] R. Rivest and R. Schapire. A new approach to unsupervised learning in deterministic environments. In P. Langley, editor, *Proceedings of the Fourth International Workshop on Machine Learning*, pages 364–375, Irvine, California, 1987.

[Rivest and Schapire, 1987b] Ronald Rivest and Robert Schapire. Diversity-based inference of finite automata. In *Proceedings of the Twenty Eighth Annual Symposium on Foundations of Compuer Science*, pages 78–86, 1987.

[Robinson and Fallside, 1987] T. Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department, 1987.

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

[Schaal and Atkeson, 1994] Stefan Schaal and Christopher G. Atkeson. Robot juggling: An implementation of memory-based learning. *Control Systems Magazine*, February 1994.

[Schmidhuber, 1991] Jurgen Schmidhuber. Reinforcement learning in markovian and non-markovian environments. In *Advances of Neural Information Processing Systems (NIPS 3)*, pages 300–506. Morgan Kaufmann, 1991.

[Schneider, 1994] J. Schneider. High dimension action spaces in robot skill learning. In *Twelfth National Conference on Artificial Intelligence (AAAI-94)*, 1994.

[Schneider, 1995] Jeff G. Schneider. *Robot Skill Learning Through Intelligent Experimentation.* PhD thesis, Department of Computer Science, University of Rochester, January 1995.

[Singh *et al.*, 1994] Satinder Pal Singh, Tommi Jaakkola, and Michael I. Jordan. Model-free reinforcement learning for non-markovian decision problems. In *The Proceedings of the Eleventh International Machine Learning Conference.* Morgan Kaufmann Publishers, 1994.

[Singh, 1992] Satinder P. Singh. Scaling reinforcement learning algorithms by learning variable temporal resolution models. In *The Proceedings of the Ninth International Machine Learning Conference.* Morgan Kaufmann Publishers, Inc., 1992.

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

[Sondik, 1973] E. J. Sondik. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. *Operations Research*, 26(2), 1973.

[Stanfill and Waltz, 1986] C. Stanfill and D. Waltz. Toward memory-based reasoning. *Communications of the ACM*, 29(12):1213–1228, 1986.

[Stolcke and Omohundro, 1992] Andreas Stolcke and Stephen Omohundro. Hidden Markov model induction by bayesian model merging. In *Advances of Neural Information Processing Systems 4*, November 1992.

[Sutton, 1984] Richard S. Sutton. *Temporal Credit Assignment In Reinforcement Learning*. PhD thesis, University of Massachusetts at Amherst, 1984. (Also COINS Tech Report 84-02).

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

[Sutton, 1990b] Richard S. Sutton. Integrating architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, Austin, Texas, 1990. Morgan Kaufmann.

[Sutton, 1991] Richard S. Sutton. Planning by incremental dynamic programming. In *Proceedings of the Eight International Workshop on Machine Learning*, pages 353–357. Morgan Kaufmann, 1991.

[Tan and Schlimmer, 1990] Ming Tan and Jeffrey C. Schlimmer. Two case studies in cost-sensitive concept acquisition. In *Eighth National Conference on Artificial Intelligence*, 1990.

[Tan, 1991] Ming Tan. Cost sensitive reinforcement learning for adaptive classification and control. In *AAAI*, 1991.

[Teller, 1993] Astro Teller. Genetic programming, indexed memory, the halting problem, and other curiosities. Department of Computer Science, Carnegie Mellon University (Unpublished manuscript), 1993.

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

[Tenenberg *et al.*, 1993] Josh Tenenberg, Jonas Karlsson, and Steven Whitehead. Learning via task decomposition. In *From Animals to Animats:Proceedings of the Second International Conference on Simulation of Adaptive Behavior*. MIT Press, 1993.

[Tesauro, 1994] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, acheives master-level play. *Neural Computation*, 6(2):215–219, 1994.

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

[Thrun, 1992b] Sebastian B. Thrun. The role of exploration in learning control. In *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*. Van Nostrand Reinhold, 1992.

[Ullman, 1984] Shimon Ullman. Visual routines. *Cognition*, 18:97–159, 1984. (Also in: Visual Cognition, S. Pinker ed., 1985).

[Watkins and Dayan, 1992] Christopher Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

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

[Wettschereck and Dietterich, 1994] Dietrich Wettschereck and Thomas G. Dietterich. Locally adaptive nearest neighbor algorithms. In *Advances of Neural Information Processing Systems 6*. Morgan Kaufmann Publishers, Inc., 1994.

[Whitehead and Ballard, 1991a] Steven D. Whitehead and Dana H. Ballard. Learning to perceive and act by trial and error. *Machine Learning*, 7(1):45–83, 1991. (Also Tech. Report # 331, Department of Computer Science, University of Rochester, 1990.).

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

[Whitehead *et al.*, 1993] Steven Whitehead, Jonas Karlsson, and Josh Tenenberg. Learning multiple goal behavior via task decomposition and dynamic policy merging. In S. Mahadevan and J. Connell, editors, *Robot Learning*. MIT Press, 1993.

[Whitehead, 1992] Steven D. Whitehead. *Reinforcement Learning for the Adaptive Control of Perception and Action*. PhD thesis, Department of Computer Science, University of Rochester, 1992.

[Wiering and Schmidhuber, 1996] Marco Wiering and Jurgen Schmidhuber. Solving POMDPs with Levin search and EIRA. In *The Proceedings of the Thirteenth International Machine Learning Conference*. Morgan Kaufmann Publishers, Inc., 1996.

[Wisniewski and Brown, 1993] Robert W. Wisniewski and Christopher M. Brown. Ephor, a run-time environment for parallel intelligent applications. In *Proceedings of The IEEE Workshop on Parallel and Distributed Real-Time Systems*, pages 51–60, Newport Beach, California, April 13-15, 1993.

[Wixson and Ballard, 1991] Lambert E. Wixson and Dana H. Ballard. Learning to find objects. Technical report, University of Rochester Computer Science Dept., 1991. in preparation.

[Wixson, 1991] Lambert E. Wixson. Scaling reinforcement learning techniques via modularity. In Lawrence E. Birnbaum and Gregg C. Collins, editors, *Machine Learning:Proceedings of the eighth International workshop*. Morgan Kaufman, 1991.

[Wolpert, 1995] D. H. Wolpert. Determining whether two data sets are from the same distrubution. In *The Proceedings of the Conference on Maximum Entropy and Bayesian Methods*, 1995.

[Zhang and Dietterich, 1995] W. Zhang and Thomas G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Eleventh International Joint Conference on Artificial Intelligence*, 1995.

A Kolmogorov-Smirnov Implementation

The implementation of the Kolmogorov-Smirnov test in “Recipes in C” has an error [Press *et al.*, 1988]. It does not correctly handle multiple data points with the same value.

Here is a corrected version.

```
#define EPS1 0.001
#define EPS2 1.0e-8

/* Kolmogorov-Smirnov probability function */

float
probks (float alam)
{
  int j;
  float a2, fac=2.0, sum=0.0, term, termbf=0.0;

  a2 = -2.0 * alam * alam;
  for (j = 1; j <= 100; j++)
    {
      term = fac * exp(a2 * j * j);
      sum += term;
      if (fabs(term) <= EPS1 * termbf
  || fabs(term) <= EPS2 * sum)
return sum;
      fac = -fac;
      termbf = fabs(term);
    }
  return 1.0;
}
```

```
/* Given an array data1[1..n1], and an array data2[1..n2], this
   routine returns the K-S statistic d, and the significance level
   prob for the null hypothesis that the data sets are drawn from the
   same distribution.  Small values of prob show that the cumulative
   distribution function of data1 is significantly different from that
   of data2.  Assumes that data1 and data2 are sorted. */

void
kstwo_sorted_data (float data1[], unsigned long n1, float data2[],
                   unsigned long n2, float *d, float *prob)
{
  unsigned long j1=0, j2=0;
  float d1, d2, dt, en1, en2, en, fn1=0.0, fn2=0.0;

  en1 = n1;
  en2 = n2;
  *d = 0.0;
  while (j1 < n1 && j2 < n2)
    {
      while (j1 < n1-1 && data1[j1] == data1[j1+1])
j1++;
      while (j2 < n2-1 && data2[j2] == data2[j2+1])
j2++;
      if ((d1=data1[j1]) <= (d2=data2[j2]))
fn1 = ++j1 / en1;
      if (d2 <= d1)
fn2 = ++j2 / en2;
      if ((dt=fabs(fn2-fn1)) > *d)
*d = dt;
    }
  en = sqrt(en1 * en2 / (en1 + en2));
  *prob = probks((en + 0.12 + 0.11/en) * (*d));
}
```

Index