# AUTONOMOUS DISCOVERY OF TEMPORAL ABSTRACTIONS FROM INTERACTION WITH AN ENVIRONMENT

A Dissertation Presented

by

ELIZABETH AMY MCGOVERN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2002

Department of Computer Science

**AUTONOMOUS DISCOVERY OF TEMPORAL ABSTRACTIONS FROM INTERACTION WITH AN ENVIRONMENT**

A Dissertation Presented

by

ELIZABETH AMY MCGOVERN

Approved as to style and content by:

Andrew G. Barto, Chair

Neil E. Berthier, Member

Roderic A. Grupen, Member

J. Eliot B. Moss, Member

W. Bruce Croft, Department Chair
Department of Computer Science

*This dissertation is dedicated to my parents, Bill and Gaye,*
*who have always loved and believed in me*
*and to my husband, Andy,*
*whose love and support made it possible.*

# ACKNOWLEDGMENTS

Andrew Barto has been a great thesis advisor. He has helped me to become a better researcher by shaping my critical thinking as well as by improving my expressive skills. I also benefited greatly from having Rich Sutton as my second advisor during my first two years at the University of Massachusetts. I would like to thank the members of my thesis committee, Eliot Moss, Rod Grupen, and Neil Berthier for their feedback.

Doina Precup and Kiri Wagstaff have been wonderful friends and supporters of my research. It is very helpful to have such smart women friends in CS. They provided support when I needed it and they pushed me when I needed that. I feel privileged to know Doina both as a mentor and as a friend. I thank Kiri for helpful feedback on drafts of my dissertation as well as the motivation provided by exchanging and reviewing each other's thesis chapters, as we wrote them.

I gratefully acknowledge the friendships and interactions from ALL members of the Autonomous Learning Lab (formerly known as the Adaptive Networks Lab). Beginning with my first lab meeting talk, I have received helpful feedback on the best way to present myself and my work. Robbie Moll was particularly helpful at improving my research presentations. The ongoing discussions in the lab helped to stimulate my research. I especially acknowledge Ravi, who has been a good friend and has provided many helpful comments and discussions on my research.

Mark Stehlik was the best undergraduate advisor that a person can have. He encouraged me to choose computer science and helped me to persist. Catherine Copetas also provided a tremendous amount of encouragement in my last few years at Carnegie Mellon.

I would also like to acknowledge the helpful discussions at the women's lunch gatherings with Alicia "Pippin" Wolfe, Özgür Şimşek, and Jen Neville as well as the support that

I've received from some of my friends from Carnegie Mellon. Special thanks are in order to Greg "zaz" Plesur.

Finally, I am extremely grateful for the love and support of my parents, Bill and Gaye McGovern. They taught me that I could accomplish anything that I put my mind to. They have always encouraged my interests in science and my dreams of going into space. Likewise, my wonderful husband Andrew Fagg, has provided love, support, and encouragement throughout my dissertation. I have been blessed to be surrounded by and related to them.

This work was supported by the National Physical Science Consortium, Lockheed Martin, Advanced Technology Labs, and the National Science Foundation under grant ECS-9980062 and EIA 9703217.

# ABSTRACT

## AUTONOMOUS DISCOVERY OF TEMPORAL ABSTRACTIONS FROM INTERACTION WITH AN ENVIRONMENT

MAY 2002

ELIZABETH AMY MCGOVERN

B.S., CARNEGIE MELLON UNIVERSITY

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Andrew G. Barto

The ability to create and to use abstractions in complex environments, that is, to systematically ignore irrelevant details, is a key reason that humans are effective problem solvers. Although the utility of abstraction is commonly accepted, there has been relatively little research on autonomously discovering or creating useful abstractions. A system that can create new abstractions autonomously can learn and plan in situations that its original designer was not able to anticipate.

This dissertation introduces two related methods that allow an agent to autonomously discover and create temporal abstractions from its accumulated experience with its environment. A *temporal abstraction* is an encapsulation of a complex set of actions into a single higher-level action that allows an agent to learn and plan while ignoring details that appear at finer levels of temporal resolution. The main idea of both methods is to search

for patterns that occur frequently within an agent's accumulated successful experience and that do not occur in unsuccessful experiences. These patterns are used to create the new temporal abstractions.

The two types of temporal abstractions that our methods create are 1) subgoals and closed-loop policies for achieving them, and 2) open-loop policies, or action sequences, that are useful "macros." We demonstrate the utility of both types of temporal abstractions in several simulated tasks, including two simulated mobile robot tasks. We use these tasks to demonstrate that the autonomously created temporal abstractions can both facilitate the learning of an agent within a task and can enable effective knowledge transfer to related tasks. As a larger task, we focus on the difficult problem of scheduling the assembly instructions for computers with multiple pipelines in such a manner that the reordered instructions will execute as quickly as possible. We demonstrate that the autonomously discovered action sequences can significantly improve performance of the scheduler and can enable effective knowledge transfer across similar processors.

Both methods can extract the temporal abstractions from collections of behavioral trajectories generated by different processes. In particular, we demonstrate that the methods can be effective when applied to collections generated by reinforcement learning agents, heuristic searchers, and human tele-operators.

# TABLE OF CONTENTS

# LIST OF TABLES

## LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

The ability to create and to use abstractions, that is, to systematically ignore irrelevant details in complex environments, is a key reason that humans are effective problem solvers. An abstraction can be a compact representation of the knowledge gained in one task that allows the knowledge to be re-used in related tasks. This representation enables an agent to learn and plan at a higher level. For example, if we consciously had to plan our muscle movements every time we wanted to bring a fork full of food to our mouth, the planning necessary for eating an entire meal would be nearly impossible. Instead, we create an abstraction that encapsulates the entire set of muscle movements that we use to put food onto the fork and to move it to our mouth. Using this abstraction, we can plan at a higher level that allows us to ignore details at the muscle level and to focus on larger tasks such as eating a meal and engaging in dinner conversations at the same time.

Researchers in artificial intelligence (AI) have long studied the use of abstractions to enable AI systems to solve large and complex problems. By allowing a system to ignore details that are not immediately relevant to the current task, the size of the space that an agent must search to find a solution can be reduced. One method for achieving this is to reduce the branching factor of the search by creating a hierarchy of actions. A second approach reduces the size of the state space by dropping irrelevant state variables. These approaches represent the two main categories of abstractions that have been studied: state and temporal abstractions. By *state abstraction*, we mean an abstraction that generalizes or aggregates over state variables. Using the previous example, if the muscle movements required to pick up the fork can also be applied to picking up a spoon, then the agent can generalize across the “fork” and “spoon” variables to create an abstraction of “pick up utensil.” By *temporal abstraction*, we mean an encapsulation of a complex set of actions into a single higher-level action that allows an agent to learn and plan at multiple scales in time. Temporal abstractions create a hierarchy of actions. As an example, the “pick up fork” abstraction discussed above is a temporal abstraction because a human can plan to eat using this abstraction without the need to know whether it will take one second or ten seconds to bring the fork full of food to her mouth and without the need to plan the muscle movements consciously.

Temporal abstractions are particularly useful for facilitating the control of complex systems. By allowing an agent to plan with actions that can take an extended, and possibly variable, amount of time to complete, the effective depth that the agent must search to find a solution can be shortened. This can enable an agent to solve more difficult problems. Also, by encapsulating a set of complex actions into a single higher-level action, such as the “pick up fork” example discussed above, a temporal abstraction can allow a more effective transfer of knowledge from one task to another.

This dissertation introduces two related methods that allow an agent to discover and create temporal abstractions autonomously from the agent’s accumulated experience with its environment. The temporal abstractions that our methods create make use of state abstraction, but the main focus is to create temporal abstractions. Although much of the relevant research in AI has demonstrated the utility of abstraction, there is comparatively little work on how to discover or create useful abstractions. Instead, abstractions are generally hand-crafted — a process requiring considerable time and effort. A system’s ability to create new abstractions autonomously can enable it to learn and plan in situations that its original designer was not able to anticipate.

As an example, consider a robot whose main task is to build human habitats on the moon in preparation for human arrival. To keep costs down, the robot needs to operate as efficiently and as autonomously as possible. Complete tele-operation would be expensive and difficult due to the communication time lag from Earth. Although the robot could be pre-programmed extensively on Earth, it is not possible to anticipate how every aspect of the lunar environment will differ from Earth's environment. Instead, the robot will perform best if it adapts to the new environment and learns to function as efficiently as possible. By autonomously creating new abstractions that characterize interactions with the environment, a robot will be better able to react to unknown situations. For example, if a robot has learned a temporal abstraction that encapsulates its knowledge of how to move efficiently through the lunar regolith (or soil), and it encounters an area with deeper regolith than expected, the robot should be able to use the temporal abstraction to escape more effectively. With the ability to learn and to create new abstractions, the robot will be more robust and can save both time and money for the humans involved.

We have two objectives in mind for the temporal abstractions created by our methods. The first objective is that the new temporal abstractions facilitate knowledge transfer to similar tasks. By doing so, the automatically created actions should enable an agent to solve more difficult tasks as long as the tasks have some underlying characteristics in common. The second objective applies in the cases where the abstractions are generated online from the behavior of a learning agent. In this case, we want the abstractions to accelerate the agent's learning process within the task in which these abstractions are generated. This turns out to be more difficult than the primary objective of creating abstractions for task transfer, especially on very complex tasks. This is primarily because the agent must be able to generate successful experience within the task before it can create the abstractions, which becomes more difficult as the complexity of the task increases. However, we demonstrate that it is possible on several tasks.

The main idea of both of our methods is to search for patterns that occur frequently within an agent's accumulated successful experience and that do not occur in the agent's unsuccessful experience. These patterns are used to create the new temporal abstractions.

In particular, the methods search either for "bottleneck" regions in sensory space or for frequently occurring sequences of actions.

The first type of temporal abstraction that our methods create is a closed-loop policy that achieves a subgoal. The method detects useful subgoals in the sensory space of an agent and creates policies that achieve these subgoals. These temporal abstractions take the form of a closed-loop policy, which means that the agent has a mapping from states in the environment to the actions that should be chosen to achieve the subgoal. In a closed-loop policy, the actions at any given point depend on feedback from the environment. An example of a closed-loop policy to achieve a subgoal is driving to a particular intersection. The intersection is the subgoal and the actions that the driver chooses to reach the intersection will depend on such conditions as the amount of traffic and the current weather.

The second type of temporal abstraction that our methods create is an action sequence. The methods can create two forms of action sequences, both of which are implemented as open-loop policies. An open-loop policy is one in which the actions chosen do not depend on feedback from the environment. For example, an open-loop action sequence could be "walk forward for five seconds." Once selected, the next action in the sequence does not depend on the state of the environment but only on the sequence itself. The first form of action sequence that the method can create allows the agent to selectively choose the sequence, but once the sequence has been chosen, the agent must execute each of the actions in the sequence. The second form includes the ability to conditionally terminate the sequence if the agent reaches an unexpected state. We denote the first type of sequence as an *action sequence* and the second type as a *conditionally terminating sequence*. In both case, the specific sequence of actions is fixed.

The patterns that the two methods search for in the agent's experience differ for the two types of temporal abstractions that they respectively create. To detect useful subgoals, our method looks for sensory observations that occur frequently across successful paths to a goal but not on unsuccessful ones. Bottlenecks in sensory space satisfy this criterion because the agent must pass through the bottleneck region in order for a task to be solved successfully. For both types of sequences, the methods look for sequences of actions that are chosen frequently across multiple successful paths to a goal. Unsuccessful experience is not considered when detecting these action sequences. Both of these approaches share the fundamental concept of searching in an agent's successful experience for frequent observations or actions.

We are particularly interested in using our methods with machine learning techniques that enable learning on complex tasks. Machine learning techniques provide an agent with the ability to adapt to changes in the environment as well as the ability to improve performance by learning from experience. This ability is an important component for systems, such as robots, that need to interact with the real world. Although there are successful examples of non-learning robots functioning effectively in real-world situations, such as car assembly lines, these robots are not able to operate outside of the structured environment for which they have been programmed. We are especially interested in tasks where the pre-programming of the agents or robots is very difficult, yet the agents must effectively perform in a complex and changing environment.

In particular, we focus on the machine learning approach known as reinforcement learning (Sutton and Barto, 1998). Unlike supervised learning approaches, which require an outside teacher to specify the correct actions to take at each step, a reinforcement learning agent learns directly from its interactions with its environment. Reinforcement learning has been demonstrated to be especially effective for control problems where the goal is known but the method to achieve the goal is not easily specified.

Although reinforcement learning algorithms can approximate or achieve optimal performance for many tasks, the algorithms can take impractical amounts of time to achieve this level of performance. This limits the size of feasible problems for which a reinforcement learning system can approximate solutions because it can be difficult to approximate solutions for tasks with long solution paths or large state spaces. By studying techniques to create temporal abstractions automatically, this dissertation addresses the issue of the limitations on the size of feasible problems. Current reinforcement learning systems use abstractions that are defined *a priori* to facilitate learning on larger problems. While this approach often works well, it is limited to how well the human programmers can define appropriate abstractions. By allowing abstractions to be created automatically, an agent can learn in larger problems without the need for constant human intervention.

Although the utility of temporal abstractions is commonly accepted, their use has mainly been explored in small domains. We present results where our methods are used to create temporal abstractions on a real-world instruction scheduling task. In this task, the system schedules assembly instructions in a valid manner and in such a way that the instructions will execute as quickly as possible. This is particularly useful on processors with multiple specialized pipelines in which the instructions can execute. We use this example to demonstrate the utility of temporal abstractions on a large task. This is especially relevant because computing optimal schedules is NP-complete. We also demonstrate that the automatically created temporal abstractions can facilitate knowledge transfer across similar processors.

The approach that we introduce of extracting information and patterns from the agent's accumulated experience with the environment and turning that information into temporal abstractions is novel. This approach is not limited to creating only the two types of temporal abstractions that we discuss in this dissertation. Other types of information may be able to be extracted from this data. For example, Hidden Markov Models or finite automata could both be used as another form of abstraction by an agent.

The methods that we introduce do not require that the experience used to identify and create the abstractions come only from the behavior of a learning agent. Other purposeful methods of exploring an environment, such as heuristic search or human tele-operation, can be used to generate the experience. We demonstrate that each of these various types of behaviors can be used successfully by our methods to create useful temporal abstractions.

Before describing the methods for creating temporal abstractions in more detail, we give an overview of reinforcement learning in Chapter 2. We also discuss the framework that we use to represent the temporal abstractions, that of *options* (Sutton et al., 1998, 1999; Precup, 2000). In Chapter 3, we place our work within the context of related research.

Chapter 4 presents our method for automatically discovering and creating useful temporal abstractions that achieve subgoals in sensory space. After presenting the method, we discuss experimental results in a gridworld task and in a simulated robot task. These results make use of a reinforcement learning agent to generate the behavior used to detect the subgoals, whereas, in Chapter 5, we present results with a more complicated simulated robot task that makes use of human tele-operation to generate the experience.

Chapter 6 presents our method for autonomously discovering and creating both action sequences and conditionally terminating sequences. In this chapter, we illustrate the method with experimental results in several gridworlds. Chapter 7 presents a real-world instruction scheduling task which we use as a platform for testing the sequence discovery algorithm. We present results that demonstrate both better performance for a given processor and knowledge transfer across processors when using conditionally terminating sequences. The conditionally terminating sequences in this chapter are created from the behavior of several heuristic schedulers. This chapter also explores the question of what makes some sequences more useful than others. The final chapter, Chapter 8, concludes and discusses our plans for future research in this area.

# CHAPTER 2

# FRAMEWORK AND NOTATION

This chapter gives an introduction to the reinforcement learning framework with particular attention paid to the aspects that we make use of in this dissertation. We also discuss the options framework (Sutton et al., 1998, 1999; Precup, 2000) used to represent our temporally abstract actions.

## 2.1 Reinforcement Learning

Reinforcement learning is a collection of methods for approximating optimal solutions to stochastic sequential decision problems (Sutton and Barto, 1998). An RL system does not require a teacher to specify correct actions. Instead, it tries different actions and observes their consequences to determine which actions are best. More specifically, in the RL framework, a learning *agent* interacts with an *environment* over a series of discrete time steps $t = 0, 1, 2, 3, \ldots$. At each time $t$, the agent observes the environment *state*, $s_t$, and chooses an action, $a_t$, which causes the environment to transition to a new state, $s_{t+1}$, and to reward the agent with $r_{t+1}$. In a system with a discrete number of states, $S$ is the set of states. Likewise, $A$ is the set of all possible actions and $A(s)$ is the set of actions available in state $s$. In a Markovian system, the next state and reward depend only on the preceding state and action, but they may depend on these in a stochastic manner. The objective of the agent is to learn to maximize the expected value of reward received over time. It does this by learning a (possibly stochastic) mapping from states to actions called a *policy*. More precisely, the objective is to choose each action $a_t$ so as to maximize the *expected return*,

$$\mathrm{E}\left\{\sum_{i=0}^{\infty}\gamma^i r_{t+i+1}\right\},$$

where $\gamma \in [0,1]$ is a discount-rate parameter, which allows the agent to trade off between the immediate reward and future possible rewards.

Two common solution strategies for learning an optimal policy are to approximate the *optimal value function*, $V^*$, or the *optimal action-value function*, $Q^*$. The optimal value function maps each state to the maximum expected return that can be obtained starting in that state and thereafter always taking the best actions. With the optimal value function and knowledge of the probable consequences of each action from each state, the agent can choose an optimal policy. For control problems where the consequences of each action are not necessarily known, a related strategy is to approximate $Q^*$, which maps each state and action to the maximum expected return starting from the given state, assuming that the specified action is taken, and that optimal actions are chosen thereafter. Both $V^*$ and $Q^*$ can be defined using Bellman equations as follows:

$$\begin{aligned}
V^*(s) &= \max_{a\in A}\sum_{s'\in S} P^a_{ss'}\left[R^a_{ss'} + \gamma V^*(s')\right], \text{ and} \\
Q^*(s,a) &= \sum_{s'\in S} P^a_{ss'}\left[R^a_{ss'} + \gamma \max_{a'\in A(s')} Q^*(s',a')\right].
\end{aligned}$$

$P^a_{ss'}$ is the probability of transitioning from state $s$ to state $s'$ under action $a$ and $R^a_{ss'}$ is the expected reward for taking action $a$ in state $s$ and transitioning to state $s'$. We use the control strategy of learning to approximate $Q^*$ for the experiments in this dissertation by using Q-learning (Watkins, 1989) as the basis for learning the action-value pairs. The update rule for Q-learning is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r_t + \gamma \max_{a'} Q(s',a') - Q(s,a)\right], \tag{2.1}$$

where $\alpha \in (0,1]$ is a step-size parameter which controls how quickly $Q$ changes at each update. Each update is also called a backup. See Sutton and Barto (1998), Sutton (1988), and Watkins (1989) for related learning algorithms.

We define some additional terminology and assumptions for this dissertation. Instead of assuming that the agent observes its state as a single member of a set, we assume that the agent is using a factored state representation (Dean and Lin, 1995; Boutilier et al., 1995, 1999). This means that the agent's perceived state is composed of a vector of sensory information. For example, using a factored representation in a gridworld, the agent could observe its $x$ and $y$ coordinates instead of simply observing its grid square. Depending on what sensory readings are available to the agent, the complete state may not be accessible. If the agent has $n$ sensors, any readings from a non-empty subset of those $n$ sensors will be called a *sensation*.

As the agent interacts with the environment, it observes a sequence of states (and sensations), actions, and rewards. This sequence of experiences can be saved and denoted as a *trajectory*. More formally,

> **Definition:** A *full trajectory* is a sequence of states, actions, and rewards $s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, r_{t+2}, \ldots, s_{t+n}, a_{t+n}, r_{t+n+1}$ where $t$ is a time step ($t = 0, 1, 2, \ldots$) and $n$ is a non-negative integer.

The agent can then observe each of the different parts of the trajectory separately by examining only the states, actions, or rewards.

> **Definition:** A *reward trajectory* is a sequence of rewards $r_{t+1}, r_{t+2}, \ldots, r_{t+n+1}$ where $t$ is a time step ($t = 0, 1, 2, \ldots$) and $n$ is a non-negative integer.

> **Definition:** A *state trajectory* is a sequence of states $s_t, s_{t+1}, \ldots, s_{t+n}$ where $t$ is a time step ($t = 0, 1, 2, \ldots$) and $n$ is a non-negative integer.

> **Definition:** An *action trajectory* is a sequence of actions $a_t, a_{t+1}, \ldots, a_{t+n}$ where $t$ is a time step ($t = 0, 1, 2, \ldots$) and $n$ is a non-negative integer.

We also define an *observation trajectory* of length $n+1$ starting at time step $t$ to be the sequence of states and sensations that the agent experiences from time $t$ to time $t+n$. The action and reward trajectories can be said to end with a state $s$ if their corresponding full trajectories end at state $s$. In episodic tasks, each trajectory is composed of the experience from a single episode. For continuing tasks, a variety of methods can be used to segment experiences into finite-length trajectories. For example, a trajectory could end when a reward peak is reached as suggested by Iba's (1989) "peak-to-peak" heuristic.

## 2.2 Options

We use the options framework (Sutton et al., 1998, 1999; Precup, 2000) to define the temporally abstract actions that our agents automatically discover in this dissertation. An option is a temporally-extended action which, when selected by the agent, executes until a termination condition is satisfied. While an option is executing, actions are chosen according to the option's own policy. An option is similar to traditional open-loop macros except that instead of generating a fixed sequence of actions, it can also follow a closed-loop policy so that it can react to the environment. By augmenting the agent's set of base actions, or its *primitive actions*, with a set of options, the agent's performance can be enhanced.

More specifically, in this framework, a Markovian option is represented by the 3-tuple $\langle I, \pi, \beta \rangle$. $I$ is the option's input set, or the set of states in which the option can be initiated. The second component, $\pi$, is a closed-loop policy that is defined over all states in which the option can execute. The fact that $\pi$ is a closed-loop policy enables the option to react to changes in the environment. $\beta$ is the termination condition such that, at each state $s$, the option terminates with probability $\beta(s)$. Although the theory allows for $\beta$ to take on values other than one or zero, this is not necessary for the type of options that we use in this dissertation. The subgoal options used in this dissertation use an internal value function to generate their policy, $\pi$. This value function can be modified over time in response to the environment, which in turn changes the policy.

As an example, consider the option of walking to your car in the parking lot at work. Clearly, the input set, $I$, should only be those places from which it is viable to actually walk to your parking lot. For example, $I$ could include the building in which you work and the surrounding buildings or locations within a mile. It would not make sense to have $I$ include locations overseas or times when your car is not parked in the lot. The policy, $\pi$, would move your legs one after the other in a reasonable manner to take you to your car. If you encounter ice along the route, $\pi$ would react accordingly and change your walking stance to keep you from falling. Lastly, $\beta$ would be set to one once you have reached your car and zero everywhere else.

Sutton et al. (1998, 1999) have also extended the framework to include semi-Markovian options. Instead of requiring that $\pi$ be a Markovian policy, the policy of a semi-Markovian option can rely on the complete history since the option was initiated. We use both Markovian and semi-Markovian options in this dissertation. By using the option framework, we can rely on the theory of options which allows the use of the same learning and planning techniques as when using only primitive actions. This is accomplished by defining a corresponding Bellman equation for options. We use the learning algorithm Macro Q-learning (McGovern et al., 1997) to learn the option action-values. For primitive actions, the update rule for Macro Q-learning is the same as that for Q-learning. For options, the update is:

$$Q(s,o) \leftarrow Q(s,o) + \alpha \left[ R_o + \gamma^n \max_{a' \in A(s_{t+n})} Q(s_{t+n}, a') - Q(s,o) \right], \tag{2.2}$$

where $R_o = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{n-1} r_{t+n}$, $n$ is the number of time steps that option $o$ was executed, $s$ is the state where the option was initiated, and $s_{t+n}$ is the state where the option terminated.

The next chapter discusses related research in detail and the following chapters present our methods and experimental results.

# CHAPTER 3

# RELATED WORK

This chapter discusses related research in the AI, RL, and robotics domains. In particular, we discuss other methods for creating or using different types of abstractions and compare several specific approaches to the methods presented in this dissertation. We also include a discussion of related work on identifying sequences in the knowledge discovery and bio-informatics domains.

## 3.1 Relevant AI Literature

Both temporal and state abstractions have been widely studied since the early days of AI. Amarel's (1968) paper discussing the Missionaries and Cannibals problem was one of the first AI papers to state the need for abstraction in problem solving. Amarel presented a series of different hand-crafted abstractions for use with the Missionaries and Cannibals problem and demonstrated that the use of certain abstractions could significantly reduce the search space of the problem and thus make the problem much easier to solve. He also discussed how a macro operator, which is one form of temporal abstraction, that takes an agent to a "narrows" in a search space, can be useful for problem solving. The types of regions that he denotes as narrows are very similar to the bottleneck regions that we use to create subgoals. The narrows connect what Amarel terms "easily traversable areas." This corresponds with our intuition of bottlenecks joining different strongly connected regions of state space. In both cases, having a policy to reach the critical region, e.g. the bottleneck or narrows regions, can significantly accelerate the search for a solution by enabling an agent to move to these important regions more easily.

Much of the early research on abstraction came from joint psychology and AI research where computational systems were built to further the understanding of human problem solving. Newell et al.'s (1963) General Problem Solver and the SOAR project (Laird et al., 1986) are the two primary examples of such systems. Both of these systems use abstractions to narrow the search space of the problem solver by creating a hierarchy of sub-problems that are more easily solved. This shortens the path to a solution for the main problem which enables problem solving to work more effectively. SOAR formed its temporal abstractions by "chunking" or grouping together sequences of actions used to solve subproblems. Although these chunks helped the system to solve larger problems, SOAR sometimes suffered from the "utility problem" (Minton, 1988) where too many chunks could actually slow down the system by increasing the branching factor of the search. The work of Anzai and Simon (1979) also fits into this paradigm. They analyzed human problem solving protocols to guide them in creating a general problem solving mechanism that can create subgoals and chunk together useful sequences of actions.

We discuss these early systems for two reasons. First, it is useful to know the foundations of much of the work on abstraction in AI because we can build a more robust system by understanding the advantages and disadvantages of these approaches. Second, these systems are directly related to our work in several ways. Although Amarel (1968) does not automatically create subgoals to reach his "narrows," the concepts of narrows and bottlenecks are very similar. SOAR is most related through its ability to create useful sequences of actions, or chunks, in an autonomous manner. The chunks are generally open-loop action sequences that achieve a subgoal. Although our action sequences and conditionally terminating sequences are not specifically created to achieve a subgoal, Laird et al.'s (1986) formulation of what makes a useful sequence is similar to ours in that the sequences must be associated with success before being used to create a new action.

Other early research on temporal abstraction in AI focused on planning systems. The first systems used pre-defined temporal abstractions to facilitate planning. For example, the

STRIPS planner (Fikes et al., 1972) was one of the first planning systems to make use of temporal abstractions. Later systems focused on a more automated approach to generating planning hierarchies. For example, the ABSTRIPS system (Sacerdoti, 1974) introduced an extension of STRIPS that could automate some of the generation of planning hierarchies. Knoblock's (1990, 1991) planning system, ALPINE, automatically generated planning hierarchies by dropping literals from the goal description in an ordered manner to create a more abstract space in which to solve the problem. Prieditis (1993) introduced a system called Absolver II which automatically found admissible heuristics for use in means-ends search systems by using a set of pre-defined allowable transformations on the STRIPS goal and operator space. By generalizing the original problem formulation, Absolver II was able to significantly decrease the time needed to find a solution.

The most closely related work from the planning domain is that of Korf (1985). His system could automatically create open-loop macro operators for use in a means-ends problem solver. These macros were designed to abstract over non-serializable subgoals, or subgoals whose solutions violated a part of the overall problem that was already solved. He used the Rubik's cube as an example, where a solution path for the whole cube must contain states where earlier goals such as "solve one color" are temporarily violated. By abstracting over the non-serializable part of a process, a means-ends problem solver can solve problems that were previously intractable. A drawback of Korf's method is that it generates all possible sequences before pruning the set to those that allow an agent to abstract over non-serializable subgoals. Our method uses a more focused approach to generate action sequences.

The work discussed above on generating hierarchies and abstractions automatically while planning is related to the research presented in this dissertation mainly through the common focus on automatically generating useful abstractions. In both domains, this means creating abstractions that enable a system to be used in solving more difficult problems. However, these planning systems focused on creating abstractions in deterministic and completely observable environments. The focus in this dissertation is on control problems, which are generally neither deterministic nor completely observable. Although some of the ideas from this planning research may apply, a different approach is needed for these tasks.

Of the traditional AI search literature, Iba's (1989) paper is perhaps the most directly relevant to the methods presented in this dissertation. Although his system automatically created only open-loop macro operators in a deterministic planning domain, we use several of his ideas in the design of our subgoal and sequence creation methods. One of the most novel aspects of Iba's system, which he called MACLEARN, is that it did not just generate and test new macros blindly. MACLEARN only generated macros from examination of "successful" (and unsuccessful) trajectories, where success was defined using the heuristic evaluation function. As it searched for a path to the goal, MACLEARN saved the observed states, the selected actions, and the heuristic evaluations experienced along the way. Once the system observed that a new peak had occurred in the history of the heuristic evaluations, it evaluated the sequence of moves between the peaks for potential macro operators. This is related to our methods in that we do not blindly generate macros but instead examine "successful" data, for some task-dependent definition of success. Second, we can use his idea of searching from peak to peak to delineate trajectories in continuing tasks. For episodic tasks, a trajectory ends at a goal or failure state, but continuing tasks can be delineated using peaks in the history of rewards or evaluations that the agent received.

Each proposed new macro in Iba's system had to pass a filtering step before it could be added to the action set. The first layer of filters, denoted the *static filter*, was designed to ensure that the system did not become overburdened with macros, as discussed by Minton (1988), who called this the utility problem. The static filter also ensured that the macros satisfied any given task-specific constraints. We use the idea of a static filter in the same way to ensure that no new temporal abstractions are too similar to already existing ones and that any domain-specific constraints are satisfied. MACLEARN also contained a second filtering step, denoted the *dynamic filter*, that actively removed macros that proved to be less useful over time. This step could be important for pruning any new abstractions that prove to be less useful across multiple tasks.

## 3.2 Related Work on Detecting Sequences

The sequence detection task that we present in Chapter 6 is also related to research in knowledge discovery, combinatorial optimization, and bio-informatics. In the knowledge discovery domain, there are efficient algorithms to extract patterns or association rules from large databases. These algorithms usually seek to minimize the number of passes through the database. Related work in combinatorial optimization focuses on efficiently identifying common substrings in large collections of strings. This is related to the research in the bio-informatics domain on identifying common DNA fragments across large collections of DNA.

One example in the knowledge discovery domain is that of Oates (1999). His method extracted *distinctive sequences* from multiple sets of time series data. A distinctive sequence is one that differentiates one set of time series data from another set collected from the same source. The time series data consist of a sequence of sensory readings collected over a period of time. He used distinctive sequences to predict events that might alter the source of the time series data, such as a malfunction. This is a different focus than our work. However, one could imagine using his approach in conjunction with our methods to enable the agent to predict in additional ways.

Other work within the knowledge discovery domain, such as Oates et al. (1998) and Zaki (1998, 2001), focuses on discovering rules and structure from time series data. In these cases, the methods are discovering sequences in the data that can serve as predictive rules for the future. We discuss the most relevant aspects of Zaki's work in more detail in Chapter 6. Although these types of predictive sequences do not correspond directly to the types of sequences that we detect and create in Chapter 6, they could still be used to create useful new sequences. Rosenstein and Cohen's (1999) work on creating predictive categories, or time series clusters, from a mobile robot's sensory readings also fits into this paradigm.

A closely related task in the combinatorial optimization domain is that of identifying substrings that are common to sets of more than two strings. This task is also studied in the bio-informatics domain because it applies to identifying distinctive genetic subsequences in DNA databases. Gusfield (1997) introduces related sequence identification algorithms from both domains. The most related algorithm was introduced by Hui (1992). His algorithm identified the longest substrings that appear in at least $k$ strings in the database in $O(n)$ time, where $n$ is the total length of all the strings in the database. His solution used suffix trees to facilitate an efficient search for the substrings. The substrings that were detected by this approach are the longest substrings common to a specified number of strings. This is similar to our approach of identifying sequences that occur across a specified percentage of the agent's trajectories.

Both the bio-informatics and the combinatorial optimization fields have multiple algorithms that are related to our sequence detection task. In both fields, the efficiency of an algorithm is important because of the size of the search space. These algorithms typically require complicated data structures, such as the suffix trees mentioned above, to achieve this efficiency. It is a topic for future research to investigate how these algorithms might be adapted to the abstraction discovery problems that we consider here.

## 3.3 Relevant RL Literature

The use of both temporal and state abstraction in RL has been studied theoretically and empirically. Since this dissertation does not focus on state abstraction, we will not address the work on state abstraction directly although there has been interesting work in that area (e.g., Lin, 1993a,b; Moore, 1994; McCallum, 1995; Uther, 1998). We focus instead on RL research on temporal abstractions.

Much of the theoretical foundation for the use of temporal abstraction in RL has been provided by Precup and Sutton (1997), Parr and Russell (1997), Precup et al. (1998), Parr (1998) and Sutton et al. (1999). These researchers have shown that the use of temporal abstraction transforms a Markov Decision Process (MDP) into a Semi-Markov Decision Process (SMDP) and that convergence results still hold for the learning algorithms known to converge in the absence of abstraction. This information allows us to create and use different types of temporal abstractions with the knowledge that the learning algorithms will still converge.

We use the *options* framework (Sutton et al., 1998, 1999; Precup, 2000) to represent the temporal abstractions created by our methods. The options framework is discussed in detail in Chapter 2. Parr and Russell (1997) and Parr (1998) proposed an alternative framework where temporal abstractions are expressed as Hierarchies of Abstract Machines (HAMs) and Andre and Russell (2001) extended this to include Programmable HAMs (PHAMs). HAMs and PHAMs provide a method for easily including prior knowledge about the structure of the policy of a temporal abstraction, which can be advantageous if such knowledge is available in advance. This is not the case for the temporal abstractions that our methods create, and although the theoretical convergence results also hold for this formulation of temporal abstractions in RL, we use the options formalism.

Related theoretical work by Hauskrecht et al. (1998) involved the use of localized temporal abstractions to generate smaller and more abstract MDPs. They showed that the new MDPs could be solved more quickly than the original MDPs and that temporal abstraction helped to facilitate knowledge transfer across tasks. Their work built on the theoretical foundations described above and provided some theoretical reasons as to why temporal abstraction can be useful to an RL system.

One of the advantages to using temporal abstractions in RL is that the abstractions can enable a system to learn to solve more complex problems. Early work on scaling RL systems up to larger problems involved the use of shaping, or training the system on a series of related subproblems and then using the solutions to the subproblems to facilitate a solution for the next harder problem (Gullapalli, 1992; Singh, 1991, 1992a,c). These systems showed that shaping could successfully be used by an RL system to approximate solutions in cases where a solution would have been very difficult to find otherwise. The main limitation to these ideas is that the shaping sequence must be defined a priori by the programmer.

On the algorithmic side, Singh's (1992b, 1994) H-DYNA algorithm is an early example of using closed-loop temporal abstractions to facilitate learning in an RL framework. Other work includes Bradtke and Duff's (1995) SMDP learning algorithm, which learns action values for variable duration actions by backing up the discounted sum of the reward received while the action was executing. Macro Q-learning (McGovern et al., 1997) combines the SMDP backup for variable duration actions with the Q-learning backup for primitive actions to learn action-values for both primitive actions and options. Under appropriate conditions, this algorithm converges to an optimal policy. As discussed in Chapter 2, we use Macro Q-learning for all of the learning experiments involving options that are presented in this dissertation.

Many of the RL algorithms that use temporal abstraction create hierarchies of states or actions. The systems of Kaelbling (1993) and Dayan and Hinton (1993) are two early examples. Both of these methods create a hierarchy of value functions based on attributes of the state space and are able to accelerate learning compared to a flat, or non-hierarchical, system. This work was extended by Moore et al. (1999) to automatically generate hierarchies in goal directed systems. Ryan and Pendrith's (1998) RL-TOPs also generates hierarchies in a partially automatic manner. Their work is distinguished by the use of a traditional planning paradigm where all actions must have specified pre- and post-conditions. The work of Theocharous and Mahadevan (2002) also fits into this paradigm by demonstrating that hierarchical Partially Observable MDPs (POMDPs) can enable a robot to successfully solve more difficult tasks than are possible without a hierarchy. None of these systems adds new actions to the available action set. Rather, the systems focus on restricting the search space of the RL agent to decrease the time needed to approximate an optimal solution.

The MAXQ algorithm by Dietterich (1998) provides an alternate framework for temporal abstraction in RL. His system uses a fixed hierarchy of temporal abstractions, where each action at each level of the hierarchy has a separate value function. Each level of the hierarchy also has a value function to control the actions for that level. Using this framework, Dietterich is able to provide convergence results for MAXQ-learning. Although we use the options framework to specify the newly created temporal abstractions, the subgoal options use a separate value function to specify their policies. This is not specifically part of the the options framework but is part of the MAXQ framework. The value functions are also refined over time, as in MAXQ. MAXQ currently has no method for automatic creation of new macro-actions. Makar et al. (2001) have extended the MAXQ framework to multiple agents and further demonstrated the use of hierarchies for complex learning tasks.

Another approach in RL that is related to ours is that of Drummond (1998). He proposed a system in which an RL agent could detect both walls and doorways through the use of vision processing techniques applied to the learned value function. This enabled the agent to re-use parts of the value function for task transfer. Although his method can identify doorways and walls in a two dimensional gridworld setting, his approach was aimed at detecting sharp changes in the value function and not at identifying subgoals.

None of the work described above includes a method for automatically creating new temporal abstractions. However, three existing RL systems do address this problem. The first two systems assume that the task of the RL agent is to approximate solutions to a series of related MDPs that share a state space but have different transition probabilities. Thrun and Schwartz's (1995) SKILLS algorithm is one such system. By examining optimal policies within the set of related MDPs, SKILLS can extract a pre-specified number of action sequences that are common across the tasks. These action sequences can be useful in other tasks that share this state space. A similar system by Bernstein (1999) uses the optimal policies for a given set of tasks to generate a single new temporal abstraction that he calls a "reuse option." To do this, his method examines the probabilities of taking each action in each state for a given optimal policy. The action distribution for each state in the reuse option is then formed by averaging the action probabilities from each of the optimal policies for that state. Both of these systems suffer from their requirement to completely solve a set of related MDPs. This limits the size of the problems to those that are small enough to approximately solve in a reasonable amount of time. Both of these systems can enable transfer of knowledge among similar MDPs as well as accelerate learning on new tasks. However, neither of these goals can be accomplished using these systems unless the set of tasks share the same state set.

The third existing system that can automatically discover temporal abstractions in an RL framework is the one most closely related to approach presented in this dissertation. Digney's (1996, 1998) Nested Q-learning algorithm creates new options online while an agent is learning using Q-learning. This system creates new options by examining two criteria: frequency of state visitations and the reward gradient at each state. States that are visited frequently or states where the reward gradient is high are chosen as subgoals for new options. Like the options created in Chapter 4, these new options take the agent from any state in the environment to the subgoal. Because newer actions can call existing actions as subroutines, Nested Q-learning creates a hierarchy of actions for the agent. Our methods can also create action hierarchies.

Digney's work on Nested Q-learning shares a common goal with the work presented in this dissertation. In both of our approaches, one goal is to create temporal abstractions that will enable an agent to approximate an optimal solution more quickly than a system that uses only primitive actions. Although we share a common goal, the research presented in this dissertation and that of Digney differs in several significant ways. First, Nested Q-learning currently only works with Q-learning, whereas our methods do not require that RL be used to generate the behavior used to create the abstractions. For example, in Chapter 5, we demonstrate that the behavior of humans tele-operating a simulated robot can be used to generate subgoals. Second, our methods can create both options to reach subgoals (similar to Nested Q-learning) and useful action sequences. Another difference between our approaches is how we use the behavior of the agent to create new abstractions. Digney detects the subgoal states using raw visitation frequencies whereas we use diverse density (Maron, 1998; Maron and Lozano-Pérez, 1998) to identify the subgoals. Diverse density examines the evidence collected across multiple trials instead of within a single trial. Diverse density can incorporate negative evidence, which Digney's method is not able to do. By using diverse density to detect the subgoals, our method is also able to identify subgoals in large or continuous spaces. Digney's method relies on using histograms to count the state visitation frequencies, which limits the problems to which it can be applied.

## 3.4 Related Research in Robotics and Learning

Although industrial robots already perform complex tasks such as car assembly, these robots function in fixed situations and are unable to adapt to changes in the environment. For robots to be truly useful across a wide range of complex tasks, they must be able to learn and adapt. In order to learn and perform more complex tasks, robots must employ some form of abstraction above the level of motor torques.

One of the early methods for introducing temporal abstraction into robotics was Brooks' (1986) subsumption architecture. In this architecture, the programmer defines a layer of behaviors in which each behavior is a separate control program. The behaviors can execute simultaneously, and higher-level behaviors can access, or *subsume*, lower-level behaviors. These behaviors function as temporal abstractions for the robot by allowing the robot to abstract over variable-length control programs that accomplish a specified subgoal. This work is continued in Brooks (1990), where he introduces a new language for specifying and creating behaviors. The robots described in this later work are able to perform complex tasks with a high degree of reliability. Although he demonstrates that these behaviors can enable robots to solve more difficult tasks than before, the work is limited by the fact that the behaviors must be pre-programmed.

Mahadevan and Connell (1992) introduced a system that learns the individual behaviors using RL. Instead of requiring the designer to pre-specify each behavior, the designer only needs to specify the goal for each behavior and appropriate situations in which to use each behavior. Their system then learns a policy for each behavior through interaction with the environment by using RL. This provides a more robust set of policies than could be hard-coded by the programmer. Mahadevan and Connell demonstrate that their robot can use learned behaviors, or temporal abstractions, to accomplish difficult tasks.

A different method for using temporal abstraction in robotics was introduced by Huber et al. (1996) and Huber and Grupen (1997b,a, 1998, 1999). They introduced a control basis approach where a number of low-level controllers are constructed that solve a wide variety of tasks. Multiple controllers can execute simultaneously. These controllers can be viewed as actions that can take a variable length of time to complete. At this level, the actions are sets of simultaneously active controllers. The termination or convergence of a controller signals a new event, much like the termination of a variable length action in the SMDP framework. RL can then be used to learn different policies by viewing the controllers as temporal abstractions. They demonstrate that this approach enables robots to learn efficient policies to solve complex tasks. For example, they demonstrate that a four-legged mobile robot can learn a policy for safely and efficiently rotating in place (Huber, 2000). They also show that this rotating policy can enable the robot to accelerate its learning on the task of learning to safely walk to a target. This demonstrates that their framework can be used for knowledge transfer.

Although these systems introduce methods that facilitate the use of temporal abstraction in robotics and RL, none of these systems focus on automatically creating the temporal abstractions from interaction with the environment. Although apparently quite different from other work presented in this section, the method of Ram and Santamaría (1997) creates new temporal abstractions using case-based reasoning. Their approach extends the discrete case-based reasoning systems so that it can be applied to continuous systems, such as mobile robots. Their mobile robot was able to use case-based reasoning to create new cases for each new situation that it encountered while interacting with its environment. By doing so, their robot automatically created temporal abstractions that enabled it to solve new problems quickly and to transfer knowledge to similar tasks.

The next chapter introduces our method for identifying useful subgoals from the agent's interaction with its environment and presents empirical results in both an illustrative and more complex domain.

# CHAPTER 4

# FINDING USEFUL SUBGOALS

The first type of temporal abstraction that we focus on in this dissertation is a subgoal of achievement, which takes the form of a closed-loop policy that attains a subgoal. The particular type of subgoals that we focus on are regions in state or sensory space that are useful to reach. We first present a method by which an RL agent can discover such subgoals automatically and then illustrate the method in several gridworlds. Last, we demonstrate the utility of the approach on a simulated Pioneer II mobile robot.

The method presented in this chapter searches online for "bottlenecks" in observation space. Informally, a bottleneck is a region in the agent's observation space that the agent tends to visit frequently on successful paths to a goal but not on unsuccessful paths (for some suitable definition of success). The bottlenecks of interest are those that appear early and persist throughout learning. If the agent can discover these bottleneck regions and learn policies to reach them during the initial stages of learning, it can use these policies for more effective exploration as well as to refine its overall policy more quickly. These subgoal policies can then be used to facilitate learning in similar tasks.

We treat the problem of finding bottleneck regions as a *multiple-instance learning problem* as defined by Dietterich et al. (1997). In this type of problem, a system attempts to identify a target concept on the basis of "bags" of instances: positive bags have at least one positive instance, while negative bags consist of only negative instances. A successful trajectory corresponds to a positive bag, where the instances are the agent's observations along that trajectory. A negative bag consists of observations made over an unsuccessful trajectory. We argue that the problem of finding bottleneck regions fits this paradigm well and we use the concept of *diverse density* (Maron, 1998; Maron and Lozano-Pérez, 1998) to detect the bottleneck regions.

## 4.1 Autonomous Subgoal Discovery

The simplest approach to creating useful options is to search by generating many new options, randomly or based on simple heuristics, and letting the agent test them by adding each to its set of actions. Although some of these options may be useful, others may degrade the agent's performance, e.g., by adversely affecting its mode of exploration (McGovern, 1998b). Performance can also deteriorate due to the agent having too many actions from which to select. To help prevent this, the agent needs a more focused method for creating new options. In the approach described in this chapter, the emphasis is on discovering useful subgoals that can be defined in the agent's observation space. New options are then created to accomplish these subgoals.

To discover new subgoals, the agent searches for bottleneck regions in its observation space. The idea of looking for bottleneck regions was motivated by studying room-to-room navigation tasks where the agent should quickly discover the utility of doorways as subgoals (McGovern, 1998a). If the agent can recognize that a doorway is a kind of bottleneck by detecting that the sensation of being in the doorway always occurs somewhere along successful trajectories but not always on unsuccessful ones, then it can create an option to reach the doorway as a subgoal. This option can accelerate learning on the current task—if created early enough—as well as enable the agent to learn more rapidly on related tasks in the same or similar environments.

A more general motivation for using bottlenecks as subgoals is the effect of the options on the agent's exploration. If the agent uses some form of randomness to select exploratory primitive actions, it has a high probability of remaining within the more strongly connected regions of the state space for long periods of time. An option for achieving a bottleneck region, on the other hand, will tend to connect separate strongly connected areas by moving the agent to the gateway between the regions much more frequently. For example, in a room-to-room navigation task, navigation using primitive movement commands produces relatively strongly connected dynamics within each room but not between rooms. A doorway links two strongly connected regions. By adding an option to reach a doorway, the rooms become more closely connected. This allows the agent to more uniformly explore its environment. We have shown in previous work (McGovern, 1998b) that the effect on exploration is one of the main reasons that options are sometimes able to dramatically improve learning.

The idea of using bottlenecks as subgoals is not confined to gridworlds or navigation tasks. We expect that other tasks with similar weakly connected regions of state space would also benefit from subgoal discovery as described in this chapter. For example, consider a game in which the agent must find a key to open a door before it can proceed. If it can discover that having a key is a useful subgoal, then it will more quickly learn how to advance from level to level. Although our approach may be widely applicable, it clearly will not work for every task since goals of achievement are not useful in all environments and not all tasks exhibit the requisite kind of dynamics. Next, we describe a method for defining and detecting such bottleneck regions.

### 4.1.1 Multiple-Instance Learning and Diverse Density

Multiple-instance learning problems as described by Dietterich et al. (1997) are supervised learning problems in which each object to be classified is represented by a set of feature vectors, where only one of the feature vectors may be responsible for the object's observed classification. An example from explanation-based learning that these authors give is a case in which many explanations for an observed result can be obtained from a domain theory, but only one explanation can account for all the observed results. More specifically, there are multiple positive and negative bags of instances. Each positive bag must contain at least one positive instance from the target concept but may contain many negative instances. Each negative bag must contain only negative instances. The individual instances within each bag are not labeled with respect to the target concept. The goal is to learn the concept from the evidence presented by the different bags.

The problem of mining collections of trajectories for bottlenecks, or other concepts useful for defining subgoals, can be formulated as a multiple-instance learning problem. Each trajectory can be viewed as a bag, with the agent's individual observation vectors being the instances within the bag. A positive bag contains observation vectors from a successful trajectory; a negative bag contains the observation vectors from an unsuccessful trajectory. What constitutes a successful or unsuccessful trajectory can be defined in a problem-dependent way. For example, successful trajectories might be all those trajectories in which the agent reached a goal state independent of how many steps that the agent took to reach the goal. Alternatively, success might depend on reaching a goal within a certain number of steps. A bottleneck region of observation space as described above corresponds to a target concept in this multiple-instance learning problem: the agent experiences this region *somewhere* on successful trajectories and not on unsuccessful trajectories.

Maron (1998) and Maron and Lozano-Pérez (1998) devised the concept of diverse density to solve multiple-instance learning problems. The most "diversely dense" region in feature space is the region with instances from the greatest number of positive bags and the smallest number of negative bags. This differs from the concept of simple density by including the idea of using many different bags rather than one large set of instances. The concept of a diversely dense region corresponds exactly to our concept of a bottleneck region. A region with high diverse density will be a bottleneck region which the agent passes through on multiple successful trajectories and not on unsuccessful ones.

To describe how to apply diverse density to subgoal discovery, we first give an overview of Maron's definitions and derivations. Maron defines the diverse density of a target concept, $c_t$, to be:

$$DD(c_t) = Pr(c_t|B_1^+,\ldots,B_n^+,B_1^-,\ldots,B_m^-), \tag{4.1}$$

where $Pr(c_t)$ is the probability that $c_t$ is the correct concept, $B_i^+$ is the $i^{th}$ positive bag, and $B_i^-$ is the $i^{th}$ negative bag. The concept with the maximum DD value is the output of a DD search. To perform this search, we must expand Equation 4.1. Using Bayes' Rule, Equation 4.1 can be rewritten as:

$$DD(c_t) = \frac{Pr(B_1^+,\ldots,B_n^+,B_1^-,\ldots,B_m^-|c_t)Pr(c_t)}{Pr(B_1^+,\ldots,B_n^+,B_1^-,\ldots,B_m^-)}. \tag{4.2}$$

Assuming a uniform prior probability over the target concepts and noticing that the denominator is constant with respect to the target concept, finding the concept with the maximum DD value reduces to finding the maximum likelihood of the particular set of positive and negative bags given a specific concept:

$$Pr(B_1^+,\ldots,B_n^+,B_1^-,\ldots,B_m^-|c_t).$$

Maron assumes that the bags are conditionally independent given the target concept, which allows the likelihood to be rewritten as:

$$\prod_{1\leq i\leq n} Pr(B_i^+|c_t) \prod_{1\leq i\leq m} Pr(B_i^-|c_t). \tag{4.3}$$

However, it is still not possible to calculate this exactly without a model of how the bags were generated. Instead, one can use Bayes' Rule again to rewrite Expression 4.3 as:

$$\prod_{1\leq i\leq n} \frac{Pr(c_t|B_i^+)Pr(B_i^+)}{Pr(c_t)} \prod_{1\leq i\leq m} \frac{Pr(c_t|B_i^-)Pr(B_i^-)}{Pr(c_t)}. \tag{4.4}$$

Substituting Expression 4.4 into Equation 4.2, and taking into account the terms that do not depend on $c_t$, the task reduces to that of finding the maximum likelihood over concepts as follows:

$$\prod_{1 \leq i \leq n} Pr(c_t|B_i^+) \prod_{1 \leq i \leq m} Pr(c_t|B_i^-).$$

It remains to determine the probability of an instance in a bag causing the concept to be correct, $Pr(c_t|B_i^+)$ and $Pr(c_t|B_i^-)$. Maron discusses several ways to do this. We follow his suggestion of using a noisy-or model (Pearl, 1988), in which case we have:

$$Pr(c_t|B_i^+) = 1 - \prod_{1 \leq j \leq p} (1 - Pr(B_{ij}^+ \in c_t)), \text{ and} \quad (4.5)$$

$$Pr(c_t|B_i^-) = \prod_{1 \leq j \leq p} (1 - Pr(B_{ij}^- \in c_t)), \quad (4.6)$$

where $B_{ij}$ is the $j^{th}$ instance of the $i^{th}$ bag and $p$ is the number of instances in bag $B_i$.

The only part of Equations 4.5 and 4.6 that is undefined is the probability of a particular instance belonging to the target concept: $Pr(B_{ij} \in c_t)$. This can be defined in several ways. For the experiments in this chapter, we used Maron's single point concept class, although we have also experimented with linearly separable concept classes (McGovern and Barto, 2001). Maron defines the single point concept class using a Gaussian probability distribution, where the probability of a particular instance in feature space belonging to a point representing the concept falls off exponentially with the distance from the point. The concept, $c_t$, is assumed to be a $k$ dimensional vector and the probability of an instance belong to the concept class is:

$$Pr(B_{ij} \in c_t) = \frac{\exp\left(\frac{-\sum_{1 \leq l \leq k}\left(B_{ijl} - c_{tl}\right)^2}{\sigma^2}\right)}{Z}, \quad (4.7)$$

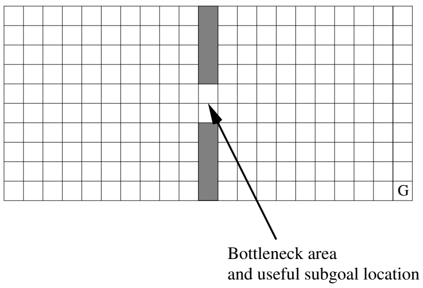
where $B_{ijl}$ is the $l$th feature of the $j$th instance of the $i$th bag, $c_{tl}$ is the $l$th feature of concept $c_t$, $Z$ is a scaling factor, and $\sigma^2$ is the standard deviation. The standard deviations $\sigma_+^2$ and $\sigma_-^2$

can be chosen separately for positive and negative bags to allow the two types of evidence to have different amounts of influence on the DD values.

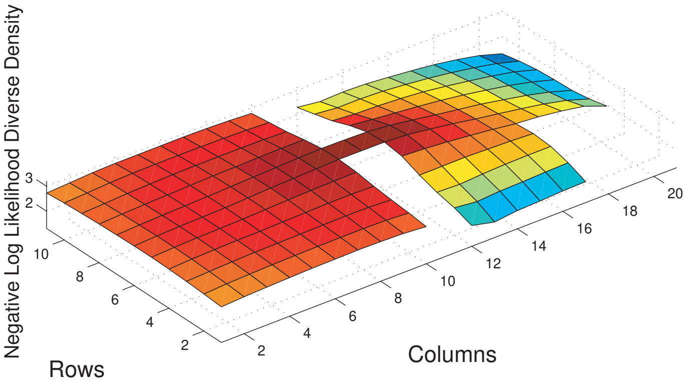
The point with the maximum DD value can be found using standard search techniques. We use exhaustive search when it is feasible and otherwise use a simple random search, which is similar to a genetic algorithms approach, as described by Rosenstein and Barto (2001). When the concept space is continuous, there are an infinite number of potential concepts. In this case, we use the search heuristic suggested by Maron where the search starts multiple times, each at randomly chosen instance from a positive bag. The true target concept should occur within multiple positive bags, which means that this approach is likely to choose several starting concepts that are near the target concept. In practice, it is more accurate to calculate the log likelihood of the diverse density, rather than calculating the diverse density directly, because of accuracy issues with very small floating point numbers.

Diverse density is similar to a computer vision technique for identifying curves and edges in images that was introduced by Hough (1962). This technique can be extended to identify general shapes in images (Ballard and Brown, 1982). The method works by first identifying a shape to be detected in an image. The location of the shape in the image can be described by a set of parameters, for example a line can be described by its slope and intercept points. Although these parameters are usually real-valued, the probable values for each parameter can be discretized into a set of bins. The Hough method initializes a count for each of the possible parameter values. As the image is examined, each pixel provides evidence for some values of each parameter. This evidence is accumulated in the count for each value. For example, if the method is searching for a straight line in an image, any pixel in the image can be used to calculate a slope and intercept for a line passing through that pixel. After the entire image has been examined, the parameters with maximal counts are chosen as the correct parameters.

The Hough method is similar to diverse density in that diverse density also specifies a parameterized concept space and accumulates evidence for specific parameters using the




A




B

**Figure 4.1.** A: The two room gridworld used for the initial experiments. B: The average negative log likelihood diverse density in a two-room gridworld shows a peak in the doorway.

bags. Unlike the Hough method, diverse density can also make use of negative evidence. One drawback to the Hough method is that it does not scale well as the number of parameters increases. Diverse density uses a more general approach of probabilities instead of discrete counters for each parameter, which enables it to be used in larger spaces. Another difference is that diverse density looks for evidence across different bags while the evidence used by the Hough method is assumed to be from one source.

We illustrate the ability of diverse density to highlight a bottleneck region using the online experience of an RL agent learning in the two-room gridworld shown in Figure 4.1A. The bottlenecks should appear as peaks in the negative log likelihood diverse density values when calculated over the entire state space. In this example, there should be a peak in the values in the doorway between the two rooms.

We used Maron's single-point concept class where each gridworld square is considered to be a target concept. We implemented this by placing the center of the Gaussian probability distribution (Equation 4.7) in the center of each grid square, and setting the standard deviation, $\sigma^2$, to 1.0.

The data used to create the bags were collected from the online behavior of an agent using Q-learning in the 21x10 two-room gridworld. The goal state was placed in the lower right-hand corner, and each episode started from a randomly chosen state in the left-hand room. An episode ended when the agent reached the goal. The primitive actions available to the agent were `up`, `down`, `right`, and `left`. Each action succeeded in moving the agent in the chosen direction with probability 0.9 and in a uniform random direction with probability 0.1. The agent received a reward of 1 for reaching the goal and 0 otherwise. The discount factor, $\gamma$, was 0.9, which made shorter trajectories to the goal better than longer ones. The learning rate, $\alpha$, was 0.05. For deterministic environments, $\alpha$ can be set to one. However, to allow convergence in stochastic environments, we usually chose a learning rate closer to zero, such as 0.05. The agent explored using the $\varepsilon$-greedy method where the greedy action was selected with probability $(1-\varepsilon)$ and a random action was selected otherwise. We set $\varepsilon$ to 0.1 for this example.

In this example, all bags were positive since the agent always succeeded in reaching the goal. Each observation, in this case the gridworld state, was added to the bag corresponding to that trajectory except for the observations within a small radius of the goal. These were excluded by a *static filter*, which is discussed in more detail in the next section.

Figure 4.1B shows the negative log likelihood diverse density value for each state in the gridworld except those surrounding the goal. The walls are blank because they are never experienced and thus have undefined diverse density values. The squares in the graph correspond to squares in the gridworld. Each data point is the negative log likelihood of the diverse density after 40 episodes of learning and averaged over 30 runs. Each run started learning from scratch with a different random seed. As the graph shows, there is a peak in the negative log likelihood diverse density in the doorway of the gridworld, which was the result that we anticipated.

```
Initialize full trajectory database to ∅
Initialize running averages ρ_c to 0
For each episode
    Interact with environment/Learn using RL
    Add observed full trajectory to database
    Create positive or negative bag from filtered state trajectory
    Search for diverse density peaks
    For each peak concept c found
        Update the running average by ρ_c ← ρ_c + 1
        If ρ_c is above threshold θ
            If c passes static filter
                Initialize I by examining trajectory database
                Set β(c) = 1, β(S − I) = 1, β(·) = 0 else
                Initialize policy π using experience replay
                Create a new option o =< I, π, β > to reach concept c
    Decay all running averages by ρ_c ← λρ_c
```

**Table 4.1.** Pseudocode for the subgoal-creation algorithm.

### 4.1.2 Using Diverse Density to Form New Options

An RL agent can use diverse density to search for the bottleneck observations, and then it can use those bottlenecks to create new subgoals and options to reach the subgoals. The algorithm for doing so is summarized in Table 4.1 and discussed in detail in this section.

At the end of each episode, the agent saves all of its observations and actions from that episode. The agent then creates a bag for the diverse density calculations consisting of the observation vectors of that trajectory. The decision whether to label the bag as positive or negative is based on whether the episode was "successful" according to the task's definition of success. In most cases, success means that the agent achieved the goal of the task within a predefined number of time steps or that it did not visit any failure states (such as hitting a wall in a navigation task).

A task-dependent filter can be used at this step to remove observation vectors from bags. For example, if the agent always starts or ends each successful trajectory with the same set of observations, then those observations will have a high diverse density since each will appear in all of the positive bags. A filter can exclude observations surrounding the start and end of each episode from any bag since these observations are not useful subgoals but would show up as peaks in the diverse density. This follows Iba's (1989) use of static filters for macro-growing. A static filter can be defined on a per-task basis and used to filter out any undesired subgoals before they can be created by the agent.

Once a set of bags has been created, a search is conducted for concepts with globally maximum diverse density values. In the initial stages of learning, there will be only a small number of bags formed from the agent's early exploratory experience. While this early experience may contain bottlenecks, it will also contain a number of spurious observations, e.g., getting stuck near a wall, which can cause the initial results of a diverse density search to be highly variable. As the agent gains experience and additional bags are created, the diverse density values should stabilize. However, as the agent's policy continues to improve, the diverse density values of the observation vectors along the optimal trajectory will begin to dominate the other values. The bottlenecks of interest are those that appear early (within the initially noisy experience) and persist throughout learning. Such early and persistent observation vectors should be indicative of general features of the environment rather than part of the specifics of a particular task because they appear within the exploratory experience as well as within the learned behavior.

One method for detecting early and persistent maxima is to maintain a running average of how often each concept is found to be have a maximal DD value. After each search, the running average, $\rho_c$, for each concept $c$ is decayed by a fraction $\lambda$, and the running averages for the concepts found to be maximal are incremented by one; that is:

$$\rho_c \leftarrow \begin{cases} \lambda\rho_c + 1 & \text{if } c \text{ is found to be maximal} \\ \lambda\rho_c & \text{otherwise} \end{cases} \tag{4.8}$$

where $\lambda \in (0,1)$. If a concept is an early and persistent maximum, then $\rho_c$ will be incremented at the end of each search. In this case, $\rho_c$ will converge toward $\frac{1}{1-\lambda}$. The agent examines the concepts whose averages rise above a specified threshold, $\theta$, and uses those to create new options. Knowing the limiting value of $\rho_c$ helps to pick the appropriate value for $\theta$.

Once a concept's average rises above the threshold, the concept is passed to the static filter. As discussed above, this filter allows the designer to restrict the set of new options based on specifics of each environment. We also use this filtering step to maintain a minimum distance between new options. Too many actions can slow learning by creating a larger search space for the agent. With an appropriate distance metric for the options, the static filter can be used to keep the agent from creating too many similar options. For example, we use the Manhattan distance between the subgoal locations for the gridworld experiments.

Once a concept has successfully passed the filtering step, it is used to create a new option. The option's input set, $I$, can be initialized in several ways, depending on the amount of knowledge that the agent has at the time the option is created. The method that we used to obtain the results reported in this chapter is to search the observation trajectories for occurrences of the subgoal. When the subgoal is found in a trajectory at time step $t$, the set of observations visited by the agent from time $t-n$ to $t$, where $n$ is a positive integer specified as a parameter, is added to $I$. The termination condition of the new option, $\beta$, is set to one for subgoal observation and for all observation vectors in $S-I$, and is set to zero otherwise. This means that the option executes either until the subgoal is achieved or until the agent exits the input set.

The option's policy, $\pi$, is initialized by creating a new value function that uses the same state representation as the overall problem. The option's value function is learned using Lin's (1992) experience replay method with the saved trajectories. The reward used for the new option is $-1$ per step and zero when the option terminates. We selected this reward structure to encourage the agent to reach the subgoal as quickly as possible. The agent could also be rewarded negatively for leaving the input set.

Saving the full history of the agent and using experience replay introduce questions about the efficiency of our method in terms of both time and space. Although saving the full history seems expensive in terms of space, it is linear in the length of the trajectories. If memory is at a premium, the agent could save only the last $n$ trajectories. This has not been a problem in practice over a number of different tasks including the simulated robot task described below. We examine the effect of the time spent in experience replay in later experiments.

## 4.2 Experimental Results

To illustrate the subgoal discovery method, we describe results for two very simple gridworld problems. We then describe results using a more realistic problem involving a simulated Pioneer II mobile robot.

### 4.2.1 Two-Room Gridworld Illustration

The first gridworld is the two-room environment shown in Figure 4.1A. The environment and learning parameters are the same as described in Section 4.1.1. In this environment, the agent's task is to move as quickly as possible from a random state in the left-hand room to the lower right corner of the right-hand room. This task allows us to examine the subgoals that the agent learns, how the newly created options affect learning, the complexity of using experience replay, why the newly created options prove to be useful, and how the options facilitate task transfer. We measured performance in each of the experiments by the number of steps that the agent took to reach the goal on each episode.

The agent created a positive bag for each trajectory in which it successfully reached the goal state from the start state. Because the agent always reached the goal state, no negative bags were created. The observations of the agent correspond to the gridworld states. We were able to calculate the diverse density exactly for each state because of the small number of states. The agent learned using Macro Q-learning. The internal value

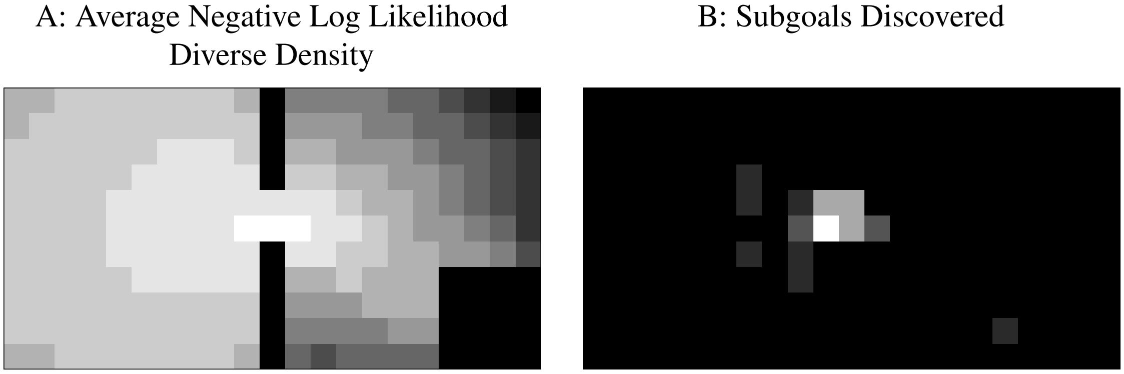



**Figure 4.2.** A: Average negative log likelihood DD values where the states with higher negative log likelihood DD values are brighter. The static filter excluded the goal region while the walls were never reached and therefore have undefined values. B: Locations of the new subgoals formed by the learning agent. Each state is shaded by the number of times that it was selected as a subgoal location; brighter squares were selected more often.

function of the option was updated using Q-learning. We limited the agent to creating only one option per run because we knew a priori that there was only one bottleneck region in the environment. Each run started learning from scratch with a different random seed. The decay factor, $\lambda$, for the running average, $\rho$, was 0.9 and the threshold, $\theta$, was 8.0.

Figure 4.2A shows the average negative log likelihood of the diverse density for each state after 25 episodes. The results in this graph are averaged over 30 runs. Each square indicates the negative log likelihood diverse density value for the corresponding gridworld square where higher values are brighter. Although this average graph is slightly smoother than the individual graphs for each run, the peak in diverse density near the doorway remains the same. These results parallel those shown in Section 4.1.1. Figure 4.2B shows the locations of the subgoals created over the 30 runs. The brighter squares were chosen more frequently. As expected, locations near the door were the most frequently detected subgoal locations.

One reason that it is important for the learning agent to detect these bottleneck states is the effect on the rate of policy convergence. If the discovered subgoals are useful, then learning should be accelerated. To determine whether these subgoals were helping the

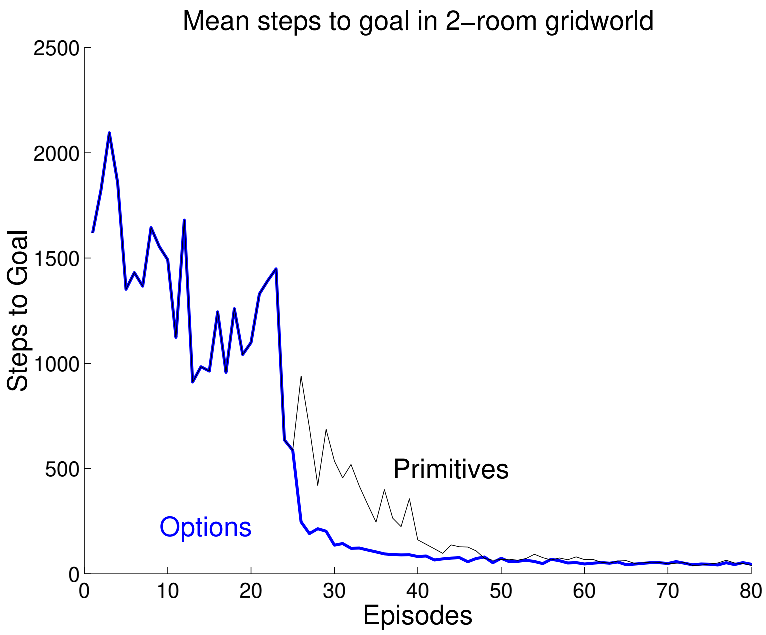



**Figure 4.3.** Average steps to the goal with and without automatic subgoal detection in the two-room gridworld. The results are averaged over 30 runs.

agent to improve its policy more quickly, we compared the average performance of an agent using subgoal discovery to the performance of an agent learning using only primitive actions. The data were averaged over 30 runs. The results of this comparison are shown in Figure 4.3. The agent using option discovery uses significantly fewer steps to achieve an optimal policy. Learning with automatic subgoal discovery has considerably accelerated learning compared to learning with primitive actions alone. Although the variances are not shown on the graph, the difference in variance between each data point on the two learning curves is greater than one standard deviation. The initial episodes were the same because the options were not created until approximately episode 20. This initial time can be modified by changing $\theta$. We found in practice that as the threshold was set closer to $\frac{1}{1-\lambda}$, the agent created better options. There is a tradeoff between how quickly the options can be created and how valuable the options are in the overall task and for task transfer.

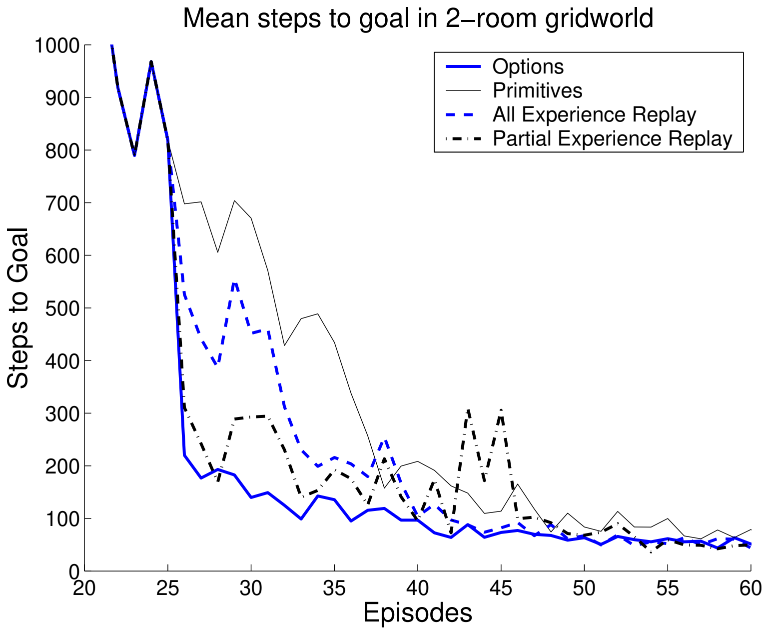



**Figure 4.4.** Results of using experience replay to examine the effect of the option's initial policy (partial-experience-replay) and to asses the time spent in experience replay while creating options (all-experience-replay). The results are averaged over 30 runs.

Future work addresses a dynamic filter that could prune options from the action set if they proved to be less useful than expected, which could be used to address this tradeoff.

These results indicate that the automatically discovered options can be useful for accelerating learning within a given task. To more fully understand why these options proved to be useful, we performed an additional experiment to examine the effect of the option's initial policy on learning. In our framework, each option's policy is initialized using experience replay. This means that once the option is created, the agent immediately has a multi-step action with a reasonable policy to move to the subgoal. This will considerably affect the agent's exploration. We wanted to examine the effect of this initial policy separately from any effect of adding an additional action to the action set. To do so, we allowed the agent to run the option discovery algorithm to detect subgoals. However, instead of creating a new option, the agent only calculated the states that would be updated by experience replay while creating the policy for the new option. It then used experience replay to backup values in the main value function on these states. This gave the agent knowledge of how to navigate to the doorway from those states.

The results of this experiment are shown in Figure 4.4 and are labeled as "Partial Experience Replay." The option-discovery and primitive action curves are the same as in Figure 4.3 but the axes have been changed to highlight the meaningful regions. The performance of the agent in this experiment is better than that of the agent using primitive actions but slightly worse than that of the agent using option discovery. The graph shows that the performance of the agent using this partial experience replay was almost as good as the performance of the agent using option discovery. This demonstrates that the primary reason newly created options are beneficial is their initial policy. The remainder of the difference in performance is postulated to be due to the ability of the options to propagate value information more quickly (McGovern, 1998b).

Since experience replay involves some amount of computational effort on the part of the agent, we wanted to ensure that the time spent in experience replay while initializing the option could not have been better spent by performing an equal amount of experience replay over the entire state space. To test this, we used a similar experimental setup as described in the previous paragraphs. The agent learned using the option discovery algorithm but at the point at which a new option would have been created, the agent only identified the input set and counted the number of backups that would have been used to initialize the policy. No new options were added to the action set. To fairly assess the computational effort of the experience replay when creating an option, we considered the case in which the agent spends an equal amount of time on experience replay over the entire state space. Although the agent has experience from the entire state space available, it uses only a subset of that experience when creating an option because the backups are focused on the input set. We approximately equalize the amount of time spent in experience replay by probabilistically selecting state-action-reward tuples from all of the the agent's saved experience and applying the Q-learning rule to these. These experiences span the entire

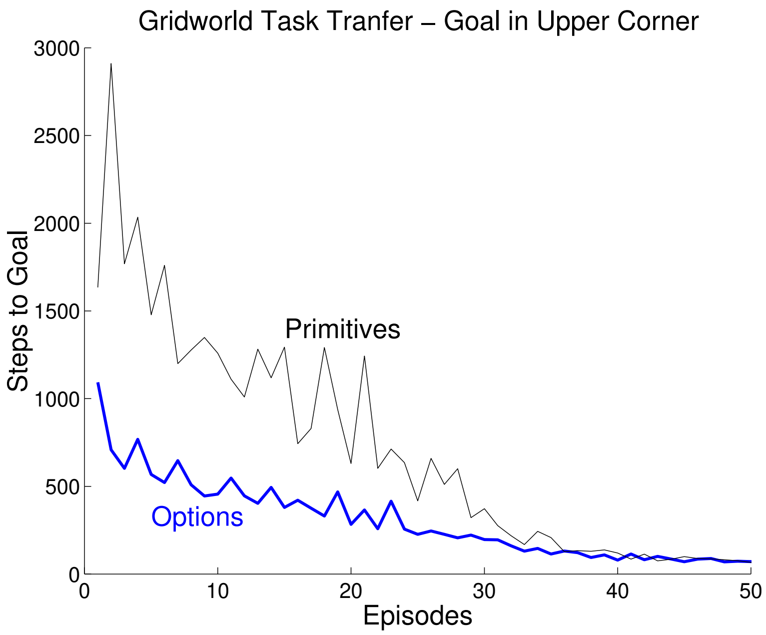



**Figure 4.5.** Learning curves in the new two-room task comparing the use of the previously learned options to primitive actions. The goal has been moved to the upper right corner and the results are averaged over 30 runs.

state space and not just what would have been the input set of the new option. Figure 4.4 also shows how the results from this case compare with earlier results. The graph for this case is labeled “All Experience Replay” and its performance is only mildly improved over that of primitive actions only. The results indicate that while experience replay can help to accelerate learning within the current task, option discovery is more advantageous.

Another reason that options can be useful is that the agent can use them to facilitate learning on similar tasks. To illustrate how learned subgoals can be useful for task transfer, the gridworld task was changed by moving the goal to the upper right-hand corner and by decreasing the probability of success for each action from 0.9 to 0.8. We reused the options discovered in the first task. In that task, we had 30 separate runs in which the agent created one option per run. In this experiment, we again have 30 separate runs, each run starting with the option discovered during the corresponding run of the previous task. We compare the performance of the agent reusing the options to an agent learning with primitive actions

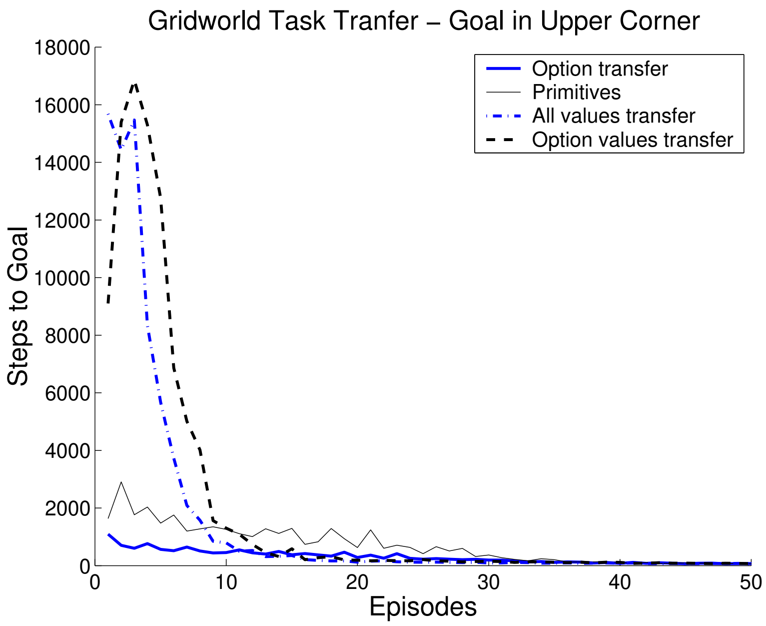



**Figure 4.6.** Comparison of knowledge transfer using the value function to using the newly discovered options. The goal is in the upper right corner and the results are averaged over 30 runs.

only. Figure 4.5 shows the results of this comparison. The agent reusing the options was able to reach the goal more quickly even in the beginning of learning and it maintained this advantage throughout. This experiment demonstrates that the availability of the options considerably accelerated learning on the new task. This was consistently observed across many different transfer tasks using this environment.

To ensure that it is the options themselves that are useful, and not just the availability of an appropriate multi-step policy in a new environment, we compared the task transfer results shown in Figure 4.5 with the results of two additional experiments. Both experiments used the value function learned in the original task, where the goal was in the lower right corner. In the first experiment, the agent initialized its entire value function using the values from this first task. In the second experiment, the agent reused only the action values from the states in the option's input set. The options themselves were not used. The results of these two experiments are shown in Figure 4.6. One can see that the agents in

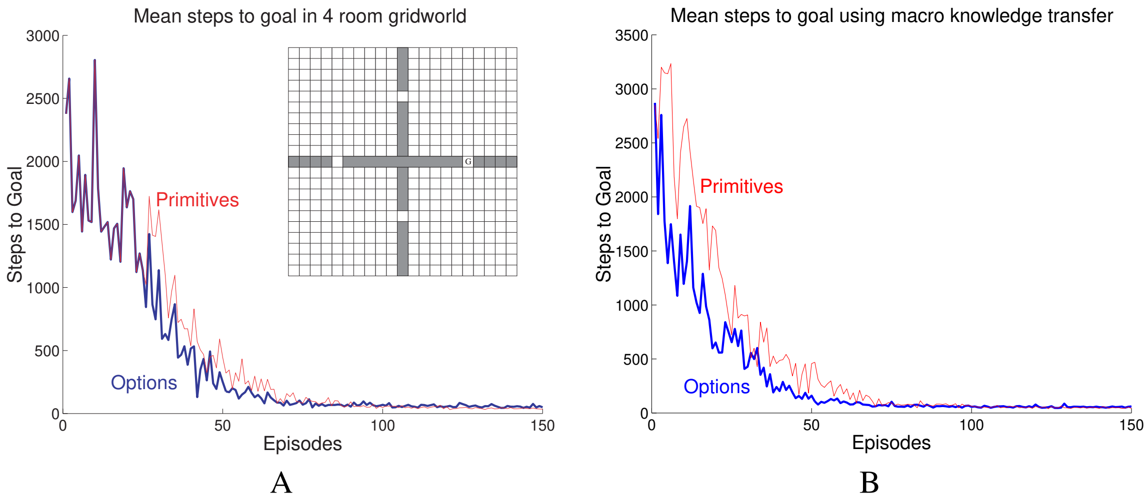



**Figure 4.7.** A: Comparison of automatic subgoal detection to primitive actions only in the four-room gridworld. B: Performance of the learned options on task-transfer in the four-room gridworld.

these two experiments took considerably longer to learn a policy to the goal than the agents starting with new value functions. These experiments demonstrate that simply reusing the value function in whole or in part is dramatically worse because the agent has to unlearn the previous task before it can learn to solve the new task.

### 4.2.2 Four-Room Gridworld Illustration

As a second simple illustration, we used the four-room gridworld studied by Precup (2000). This environment is shown in the inset of Figure 4.7A. The agent's task is to move from a randomly chosen start state in either left-hand room to the goal location in the doorway between the two right-hand rooms. The primitive actions and learning algorithms are the same as described above. The probability of success of each action is 0.9. We allowed the agent to create up to three options per run, because each non-goal doorway was a bottleneck. The static filter constrained each option's subgoal to be at least a Manhattan distance of 2 away from that of any other option.

As before, we compare the results of learning with autonomous subgoal discovery to learning with primitive actions by examining the average number of steps that the agent needed to reach the goal state from the start state. The results of this comparison are shown in Figure 4.7A. These results are averages over 30 runs. The agent using option discovery was able to use fewer steps to learn an optimal policy than the agent using only primitive actions. Although the difference between the two curves is less striking than in the two-room gridworld, here one can also see that autonomous subgoal discovery was able to improve the rate of learning. The locations of the subgoals were clustered around the 3 doorway locations.

To test knowledge transfer, we decreased the action-success probability from 0.9 to 0.8 and moved the goal to the middle of the upper-right room. Figure 4.7B compares performance with and without the previously learned options. This experiment demonstrates that the learned options facilitate knowledge transfer across these tasks.

### 4.2.3 Simulated Robot Task

As a more realistic example, we tested the subgoal discovery method presented in this chapter using a simulated Pioneer II mobile robot. The simulated environment and robot are shown in Figure 4.8. The room has dimensions of 10 units by 10 units. The robot has seven sonar sensors, a simple vision system with a 1-D "retina" having a 140° field of view, and a gripper with a sensor that signals when a ball is being gripped. The vision system was modeled as a pinhole camera. The sonars had a 20° field of view (10° on either side of nominal). The sonar reading is the distance to the closest object within that 20° field of view but it does not contain the exact heading to the object. This is consistent with the observed behavior of sonars on physical robotic platforms.

The primitive action set consists of eight actions: two distances forward (0.5 and 1.0 units), three left and three right rotations (5.625°, 11.25°, and 22.5°). Each of the primitive actions was implemented by a proportional-derivative (PD) controller that executed until it

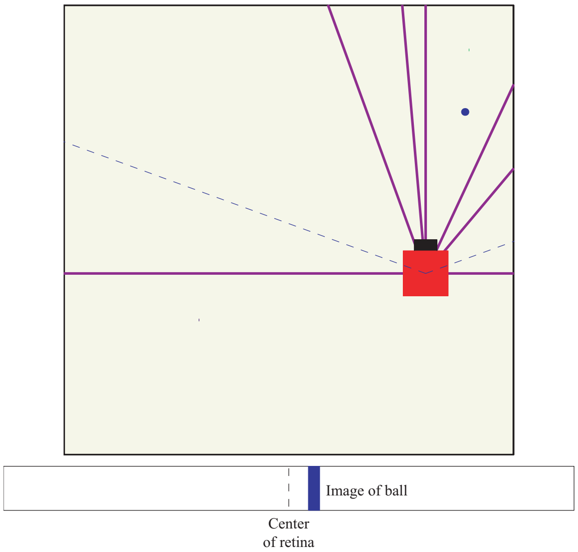



**Figure 4.8.** Simulated Pioneer mobile robot. The lines emanating from the robot represent the sonar beams and the two dashed lines show the vision cone. The rectangle on the bottom shows the ball's appearance on the retina.

stabilized the robot at the commanded setpoint (for more information about PD controllers, see Craig, 1989). Since the robot does not have access to its $x$ and $y$ coordinates, the PD controller for moving forward relies on the middle forward-facing sonar to estimate distance. The robot has an additional reflex action that was not available as an action choice: when the gripper sensor is triggered by an object coming into the gripper, the robot automatically closes and raises the gripper.

The robot's first task is to locate and pick up the ball, and then bring it within one unit of a wall without running into a wall at any time. The robot receives a reward of $+10$ for bringing the ball to the desired location, $-1$ for touching a wall, and $-0.1$ per time step until correctly solving the task. This latter negative reward encourages the robot to accomplish the task in minimum time. The discount rate, $\gamma$, was 0.995, and $\alpha$ was 0.1. The robot learned through the use of Macro Q-learning with a CMAC tile coding to represent the state space. Further information about CMAC tile codings can be found in Albus (1971) and Sutton and Barto (1998).

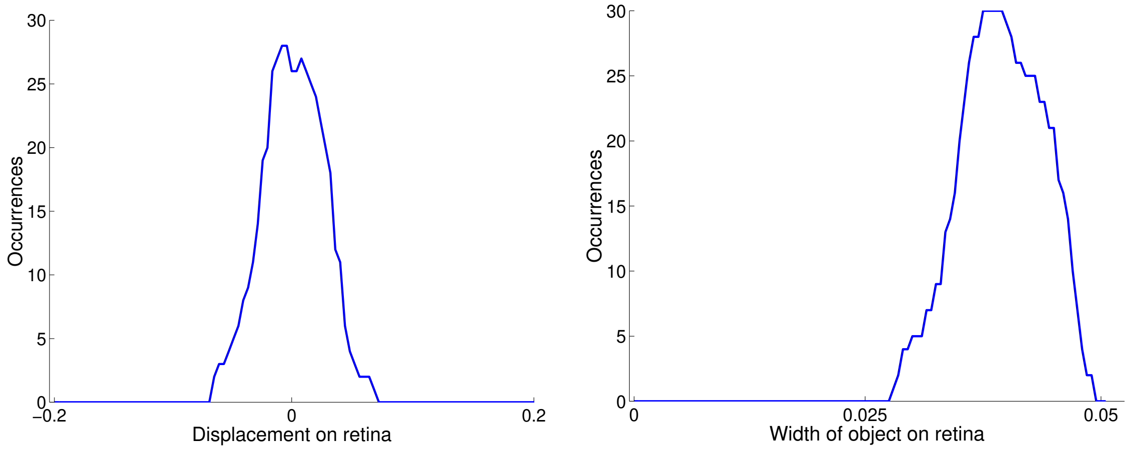



**Figure 4.9.** Frequencies of each sensory variable's occurrence in a subgoal summed over 30 runs. To achieve the subgoal, the robot must use both displacement and width of the ball's appearance on the retina.

The CMAC uses the sensory features described above including the seven sonar readings, the orientation of the object on the retina, the width of the object on the retina, and the gripper sensor. The Pioneer II robot has wheel encoders to estimate the forward and rotational velocities of the robot. Following this, the simulator is able to use the estimated forward velocity and the estimated rotational velocity in its tile coding as well. The CMAC has six layers of different subsets of these variables for a total of 2896 tiles. The details of the tiling are given in Appendix A.

The robot is able to create one option per run. The observations of the agent are the sensor readings at each time step. The subgoal is expressed using only the width and displacement of the ball's appearance on the retina. A subgoal that is useful in any task involving the robot moving the ball is for the robot to have the ball in its gripper. To do this, the ball should be looming directly in front of the robot with a large image width on the retina. Figure 4.9 shows the subgoals discovered by the robot over 30 runs as projections onto the two relevant sensory variables. To achieve the subgoal, the robot must use the conjunction of these two features. As Figure 4.9 shows, the robot discovered the subgoal of being close to the ball and being oriented directly toward it.

The results shown in Figure 4.9 demonstrate that the robot was able to extract a useful subgoal in a relatively difficult task. Because this was such a difficult task for the agent, the robot was not able to discover the subgoal in time to accelerate learning on the original task. However, we expected that the automatically created options would be useful for transferring knowledge to similar tasks. To test this hypothesis, we gave the robot a more difficult task in the same environment. Whereas the initial task required the robot to bring the ball to within one unit of any wall in the room, the new task required the robot to pickup the ball and bring it within one unit of a corner. As before, the robot was penalized for hitting a wall. This is a more difficult task because it requires the robot to better identify its location and to be able to navigate more precisely.

We compared the number of time steps that the robot took to grip the ball in this new task under two conditions: 1) when it was initialized with the options discovered in the simpler task and 2) when it did not have access to these options. The primitive actions were available in both conditions. The agent was able to create one option per run of the earlier task and the robot started with the corresponding option for each run of the new task. Figure 4.10 shows the results of this experiment. The robot reusing the options from the earlier task takes fewer steps to reach the ball than the robot learning with only primitive actions. Using the automatically discovered options significantly accelerated the initial learning of the agent. This is reflected in the number of time steps that it takes the robot to solve the overall task as well.

The results in this simulated robotic domain suggest that the method for automatic option discovery presented in this chapter is able to discover useful subgoals, and thereby facilitate knowledge transfer in non-trivial tasks, including tasks with continuous state spaces.

## 4.3 Conclusions

In this chapter, we introduced a method for automatically creating subgoal options online by searching for bottlenecks in observation space. We formulated the problem as a

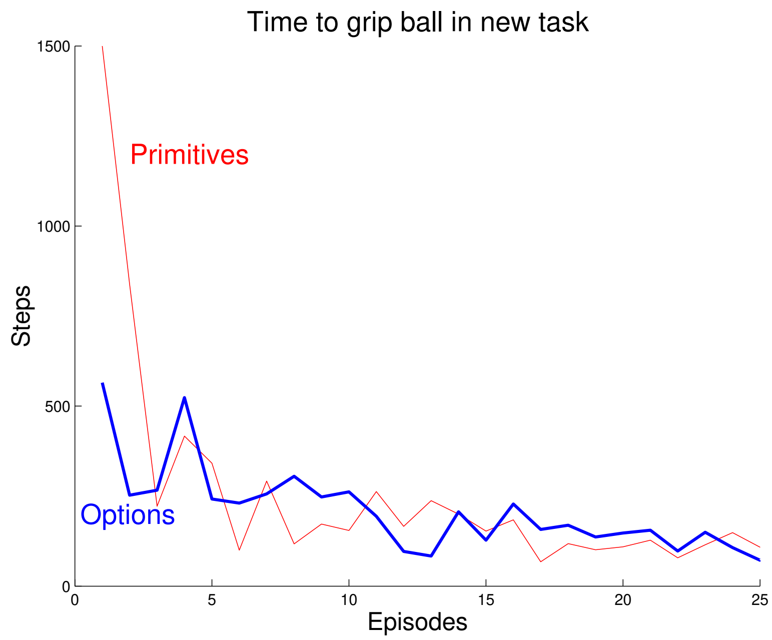



**Figure 4.10.** Results of using previously discovered options as compared to learning with primitive actions on a more difficult problem for the robot. These results are averaged over 30 runs.

multiple-instance learning problem and used the concept of diverse density to solve it. We illustrated this approach in several simulated tasks and showed that it can both accelerate learning on the current task and facilitate transfer to related tasks.

# CHAPTER 5

# GENERATING SUBGOALS FROM EXPERIENCES OF NON-REINFORCEMENT LEARNING AGENTS

The methods presented in this dissertation for generating subgoal options from the online experience of an agent do not require that the agent generate its experience using reinforcement learning. Instead, the experience can come from any purposeful method of exploring an environment. This can include other autonomous methods such as the execution of a planner or even methods such as tele-operation. In this chapter, we explore the ability of our methods to create new subgoal-based abstractions from human-generated experience. We focus on a simulated mobile robot domain.

The ability to use other forms of experience can expand the range of tasks and environments in which our methods can be used. In particular, this ability enables an agent to create useful abstractions while observing solutions for tasks which may be very difficult for current AI techniques to solve but where humans can generate useful solutions. For example, consider a tele-operated robot on the moon or Mars. The robot needs to ensure its safety while accomplishing its scientific mission. Complete tele-operation is unreliable because of the time lag in communications to and from Earth, and complete autonomy is currently too complex to achieve. Pre-programming the robot on Earth for all possible contingencies is also not possible because there are too many unknowns in how the moon or Mars differ from Earth. Instead, if an on-board learning agent could observe the mixture of actions that it generates as well as the actions generated from tele-operation, it could generate new and useful options that could enable greater independence and faster learning.

## 5.1 Experimental Setup

To more fully test the hypothesis that useful abstractions can be generated from forms of experience generated from non-RL agents, we performed several experiments in a modified version of the simulated robot environment described in Chapter 4. This environment is shown in Figure 5.1. The two-dimensional room in which the robot operated is 25 units wide by 20 units high (at the peak). The solidly shaded rectangular objects represent obstacles in the room. The task has been changed in several ways. There are now two colored balls in the room, one red and one blue. The dynamics of how each ball moves in the environment remain the same as before. The robot's new task is to locate and move the red ball to the red goal location (upper left dashed area) and to locate and move the blue ball to the blue goal location (upper right dashed area). This task is much more difficult than the one described in Chapter 4 where the robot's room contained no obstacles and the robot only had to move a single ball to be near a wall. Instead of building an autonomous system to solve the task, we created an interface that allowed a human to tele-operate the robot.

The robot used in these experiments had an augmented set of sensors compared to the robot described in Chapter 4. As before, the robot had seven sonar sensors, a simple vision system with a 1-D "retina," and a gripper with a sensor that signals when a ball is being gripped. These sensors are described in detail in Chapter 4. In addition to these sensors, the robot had two new types of sensory information available: an estimate of its position in the room and the ability to distinguish between the different colors of the balls. The latter ability adheres to the blob vision system available on Pioneer robots. Having the exact $(x, y)$ coordinates is an unrealistic assumption for a real robot due to sensor noise; instead the robot can sense an estimate of its position where the estimate is based on the robot's exact coordinates with Gaussian noise added. The Gaussian noise has a standard deviation of 0.25. With the ability to use GPS outdoors or the ability to estimate position indoors (Thrun et al., 2000), this is not an unrealistic assumption. Without some ability to know