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

© Copyright by Elizabeth Amy McGovern 2002

All Rights Reserved

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

Page

ACKNOWLEDGMENTS v

ABSTRACT vii

LIST OF TABLES xii

LIST OF FIGURES xv

CHAPTER

1. INTRODUCTION 1

2. FRAMEWORK AND NOTATION 8

2.1 Reinforcement Learning 8
2.2 Options 11

3. RELATED WORK 13

3.1 Relevant AI Literature 13
3.2 Related Work on Detecting Sequences 17
3.3 Relevant RL Literature 18
3.4 Related Research in Robotics and Learning 23

4. FINDING USEFUL SUBGOALS 26

4.1 Autonomous Subgoal Discovery 27

4.1.1 Multiple-Instance Learning and Diverse Density 28
4.1.2 Using Diverse Density to Form New Options 35

4.2 Experimental Results 38

4.2.1 Two-Room Gridworld Illustration 38
4.2.2 Four-Room Gridworld Illustration 45

4.2.3 Simulated Robot Task 46

4.3 Conclusions 49

5. GENERATING SUBGOALS FROM EXPERIENCES OF NON-REINFORCEMENT LEARNING AGENTS 51

5.1 Experimental Setup 52
5.2 Experimental Results 55

5.2.1 Visual Subgoals 56
5.2.2 Navigational Subgoals 58

5.3 Conclusions 60

6. FINDING USEFUL SEQUENCES 62

6.1 Sequence Discovery 64

6.1.1 Consec: The Sequence Detection Algorithm 66
6.1.2 Sequence Detection and Creation 70

6.2 Experimental results 73

6.2.1 Experiments with Action Sequences 73
6.2.2 Experiments with Conditionally Terminating Sequences 78

6.3 Conclusions 83

7. APPLICATION OF SEQUENCE CREATION: INSTRUCTION SCHEDULING 84

7.1 Instruction Scheduling in the Java Virtual Machine 85
7.2 Permutation Scheduling 89
7.3 Evaluating the Basic Block Schedules 90
7.4 Performance Metric 92
7.5 Benchmark Programs 93
7.6 Heuristic Schedulers 93
7.7 An RL Scheduler 95
7.8 Experimental Results 98

7.8.1 Performance of Heuristic Schedulers without Sequences 99
7.8.2 Discovering Sequences 104
7.8.3 Effect of Sequences on Exploration 109
7.8.4 Effect of Sequences on the Hill Climbing Scheduler 116
7.8.5 Knowledge Transfer Experiments 120

7.9 Conclusions 123

8. CONCLUSIONS AND FUTURE WORK 125

APPENDICES

A. DETAILS OF THE TILE-CODING REPRESENTATION USED BY THE SIMULATED ROBOT IN CHAPTER 4 129
B. DETAILED RESULTS FROM THE INSTRUCTION SCHEDULING TASK PRESENTED IN CHAPTER 7 131

B.1 Tile-coding Representation Used by the RL Scheduler 131
B.2 Conditionally Terminating Sequences Created for the Scheduler 131
B.3 Detailed Results for Each Instruction Scheduling Experiment 144

BIBLIOGRAPHY 147

LIST OF TABLES

Table **Page**

4.1 Pseudocode for the subgoal-creation algorithm. 35

5.1 Summary of the human data collected on the tele-operated robot task. 55

6.1 Pseudocode for `consec`, the sequence detection algorithm, and the method to find supported primitive actions. 67

6.2 Pseudocode for the methods used by `consec` to find all sequences of length two or more using recursive doubling. 68

6.3 Pseudocode for the sequence-creation algorithm. 70

7.1 High-level view of the permutation scheduling algorithm. 89

7.2 Features used by the RL scheduler to approximate the state space. 96

7.3 The set of derived features used to create the tile coding for the RL scheduler. 97

7.4 Performance of the BASE scheduler as compared to the original ordering for each benchmark. 100

7.5 Parameters for sequence detection and creation for each of the different data sets. 107

7.6 Results of the t-tests comparing the performance of the RANDOM scheduler with sequences to the performance of the RANDOM scheduler with no sequences. The hypothesis for each t-test is shown in the second column. Any hypothesis accepted with a p value of 0.05 or less is checked. The results for the benchmark `jack` are not shown because none of the t-tests accepted the hypothesis on this benchmark. 110

7.7 The average percent improvement toward the estimated optimal scheduler for the hill climbing scheduler using the sequences created from each of the eight data sets. The standard deviation and the minimum and maximum improvements are also given. 118

7.8 The average percent improvement toward the estimated optimal scheduler for the hill climbing scheduler on the Power PC 603 processor using the sequences created from each of the eight data sets on the Power PC 601 processor. The standard deviation, best, and worst improvements are also given. 121

A.1 Details of the simulated robot's tile-coding used to approximate the action value function. 130

B.1 Layers 1-12 of the RL scheduler's tile-coding. 132

B.2 Layers 13-21 of the RL scheduler's tile-coding. 133

B.3 Layers 22-29 of the RL scheduler's tile-coding. 134

B.4 Layers 30-36 of the RL scheduler's tile-coding. 135

B.5 Layers 37-42 of the RL scheduler's tile-coding. 136

B.6 Layers 43-48 of the RL scheduler's tile-coding. 137

B.7 Layers 49-51 of the RL scheduler's tile-coding. 138

B.8 Sequences generated from the BASE heuristic. 139

B.9 Sequences generated from the DOWNHILL heuristic. 139

B.10 Sequences generated from the HILL heuristic. 139

B.11 Sequences generated from the FARTHEST heuristic. 140

B.12 Sequences generated from the RANDOM heuristic. 141

B.13 Sequences generated from the COMPRESS heuristic. 142

B.14 Sequences generated from the RAYTRACE heuristic. 142

B.15 Sequences generated from the CROSS heuristic. 143

B.16 Performance ratios for each of the heuristics on the Power PC 601 processor. 144

B.17 Differences in the mean performance ratio of RANDOM without sequences to RANDOM with the specified set of sequences. Numbers less than 0 represent a worse performance than scheduling randomly without sequences and numbers greater than 0 represent a better performance. 145

B.18 Percent relative improvement toward an estimate of optimal of the hill climbing scheduler with sequences on the Power PC 601 processor. 145

B.19 Percent relative improvement of the hill climbing scheduler toward the estimated optimal scheduler using the sequences generated on the Power PC 601 processor on the Power PC 603 processor. 146

LIST OF FIGURES

Figure **Page**

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. 33

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. 39

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. 40

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. 41

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. 43

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. 44

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. 45

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. 47

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. 48

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. 50

5.1 The view that the human controllers see while tele-operating the simulated mobile robot. The task is to move the blue ball to the blue goal region and the red ball to the red goal region. Sensor readings which are not visually apparent to the human from the overhead view of the robot are shown on the left hand side. 53

5.2 Frequencies of each sensory variable's occurrence in a subgoal generated from the human data summed over 30 runs. To achieve the subgoal, the robot must use both displacement and width of the object's appearance on the retina. The results for the blue balls are shown with the thicker lines and the results for the red balls are shown with the thin lines. 58

5.3 Navigational subgoals discovered by the agent over 30 runs. Each subgoal location is shown as an asterisk. 60

6.1 Performance comparison between an RL agent automatically discovering new action sequences and an agent learning using only primitive actions. 75

6.2 Action sequences discovered by the agent in the first gridworld task. Each sequence is shown as a line from the origin to where the sequence would move the agent assuming that all actions in the sequence succeeded. The width and the numbers next to each line represent the number of different runs in which that the sequence was discovered. 76

6.3 Performance comparison between the agent reusing the sequences learned in the first task on a new and larger gridworld to an agent learning using only primitive actions. The results for the agent using knowledge transfer are shown with a thick line. 77

6.4 Performance comparison between the agent reusing the sequences learned in the first task on the task with the goal moved to the upper left corner (Panel A) and in the 20x20 gridworld with a wall inserted in the middle (Panel B). The knowledge transfer results are shown with a thick line. 78

6.5 A: The maze where the algorithms automatically discovered action sequences and conditionally terminating sequences. Obstacles are shown as filled in squares. B: Comparison of learning while automatically discovering the sequences to learning with primitive actions only. 80

6.6 A: The second maze world B: Comparison of learning with the automatically discovered action sequences and conditionally terminating sequences to learning with primitive actions only. 81

6.7 A: The third maze world B: Comparison of learning with the automatically discovered action sequences and conditionally terminating sequences to learning with primitive actions only. 83

7.1 Java code is transformed into machine independent byte code and then processed by the Java virtual machine, where our instruction scheduler can reorder the Power PC assembly instructions before they are transformed into machine code and executed. 87

7.2 A: A basic block of Power PC 601 instructions from the Hello World program. B: The DAG for the basic block in panel A. C: Once the instructions in bold have been scheduled, the constraints shown as dashed lines have been satisfied and any of the circled instructions can be scheduled next. 88

7.3 Comparison of the estimated best and worst running times for each benchmark. These numbers are for the Power PC 601 processor. The dashed line at 1.0 indicates a running time exactly equal to BASE. The bars above this line represent benchmarks that perform more slowly than under the BASE ordering while the bars below the line represent an ordering of a benchmark that would run faster than BASE. 101

7.4 Comparison of the FARTHEST heuristic to the RANDOM scheduler for each of the seven benchmarks. RANDOM is averaged over 30 different runs. The estimated optimal performance is also included. The thin lines on the RANDOM bars show the standard deviation above and below the mean. 103

7.5 Comparison of the HILL heuristic to the BASEHILL scheduler and the estimated optimal scheduler for each benchmark. 104

7.6 Comparison of the performance of the RANDOM scheduler with no sequences to the performance of the RANDOM scheduler with sequences created from the DOWNHILL heuristic. The error bars are one standard deviation above and below each mean. 111

7.7 Performance comparison between the RANDOM scheduler with no sequences and the RANDOM scheduler with sequences created from the FARTHEST and RANDOM heuristics. The error bars are one standard deviation above and below each mean. 112

7.8 The performance of the RANDOM scheduler using sequences created from the BASE and HILL heuristics as compared to the performance of RANDOM with no sequences. The error bars are one standard deviation above and below each mean. 113

7.9 Performance of the RANDOM scheduler as compared to the performance of the RANDOM scheduler with the sequences created from the behavior of the RL scheduler that trained on the benchmarks COMPRESS and RAYTRACE. The error bars are one standard deviation above and below each mean. 114

7.10 Performance of the RANDOM scheduler using the sequences created from the RL scheduler that trained on each benchmark (denoted CROSS) as compared to the performance of the RANDOM scheduler. The error bars are one standard deviation above and below each mean. 115

7.11 Percent improvement toward the estimated optimal schedule for the hill climbing scheduler with sequences created from the FARTHEST and RANDOM heuristics. This improvement is measured against the hill climber with no sequences. 119

7.12 Percent improvement toward the performance of the estimated optimal scheduler for the hill climber on the Power PC 603 architecture. The hill climber used the sequences generated for the Power PC 601 architecture from the behavior of the heuristics RANDOM and FARTHEST. 122

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, π, is a closed-loop policy that is defined over all states in which the option can execute. The fact that π is a closed-loop policy enables the option to react to changes in the environment. β is the termination condition such that, at each state s, the option terminates with probability $\beta(s)$. Although the theory allows for β 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, π. 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, π, 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, π would react accordingly and change your walking stance to keep you from falling. Lastly, β 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 π 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 Chap-

ter 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 mov-

ing 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 ith positive bag, and B_i^- is the ith 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:

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

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:

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

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:

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

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:

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

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

where B_{ij} is the jth instance of the ith 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:

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

where B_{ijl} is the lth feature of the jth instance of the ith bag, c_{tl} is the lth feature of concept c_t, Z is a scaling factor, and σ^2 is the standard deviation. The standard deviations σ_+^2 and σ_-^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

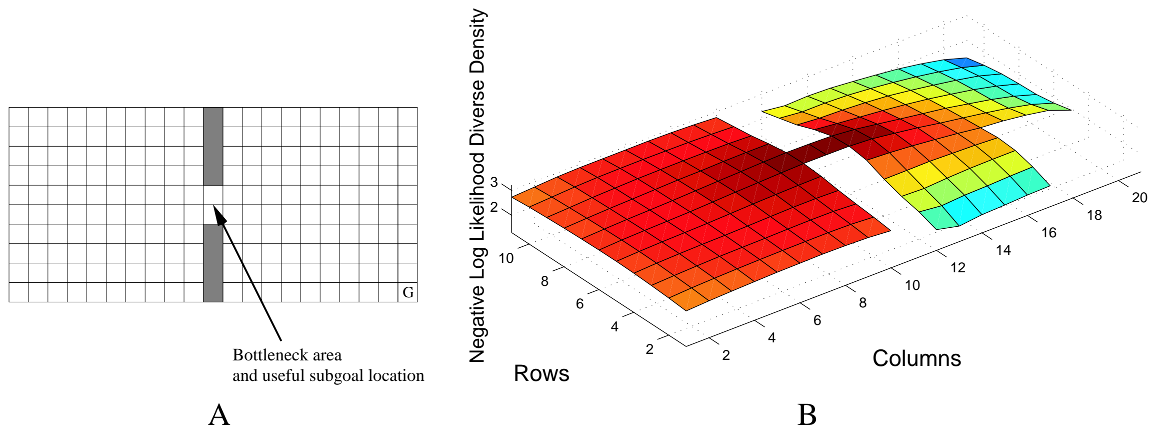


Figure 4.1. A: The two room gridworld used for the initial experiments. B: The average negative log likelihood diverse density in a two-room gridworld shows a peak in the doorway.

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, σ^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, γ, was 0.9, which made shorter trajectories to the goal better than longer ones. The learning rate, α, was 0.05. For deterministic environments, α 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 ε-greedy method where the greedy action was selected with probability $(1-\varepsilon)$ and a random action was selected otherwise. We set ε 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.

1. Initialize full trajectory database to $\emptyset$
2. Initialize running averages ρ_c to 0
3. For each episode
4. Interact with environment/Learn using RL
5. Add observed full trajectory to database
6. Create positive or negative bag from filtered state trajectory
7. Search for diverse density peaks
8. For each peak concept c found
9. Update the running average by $\rho_c \leftarrow \rho_c + 1$
10. If ρ_c is above threshold θ
11. If c passes static filter
12. Initialize I by examining trajectory database
13. Set $\beta(c) = 1$, $\beta(S - I) = 1$, $\beta(\cdot) = 0$ else
14. Initialize policy π using experience replay
15. Create a new option $o = < I, \pi, \beta >$ to reach concept c
16. Decay all running averages by $\rho_c \leftarrow \lambda \rho_c$

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, ρ_c, for each concept c is decayed by a fraction λ, 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 ρ_c will be incremented at the end of each search. In this case, ρ_c will converge toward $\frac{1}{1-\lambda}$. The agent examines the concepts whose averages rise above a specified threshold, θ, and uses those

to create new options. Knowing the limiting value of ρ_c helps to pick the appropriate value for θ.

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, β, 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, π, 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

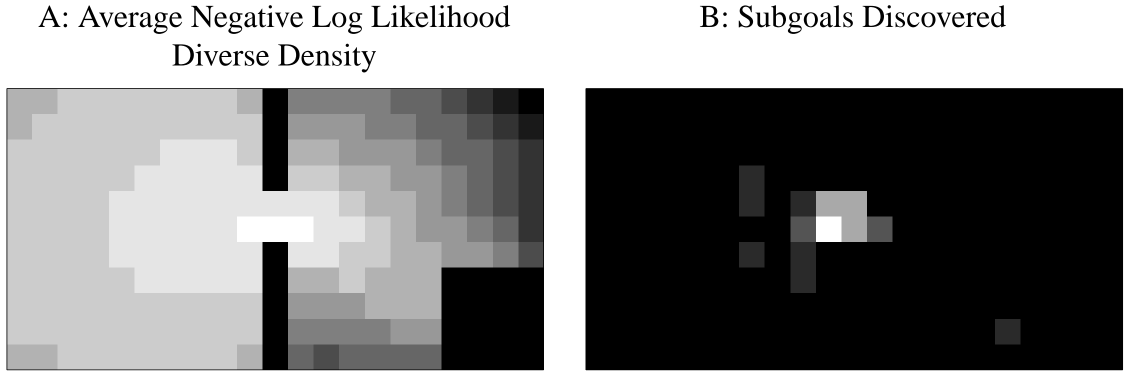


Figure 4.2. A: Average negative log likelihood DD values where the states with higher negative log likelihood DD values are brighter. The static filter excluded the goal region while the walls were never reached and therefore have undefined values. B: Locations of the new subgoals formed by the learning 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, λ, for the running average, ρ, was 0.9 and the threshold, θ, 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

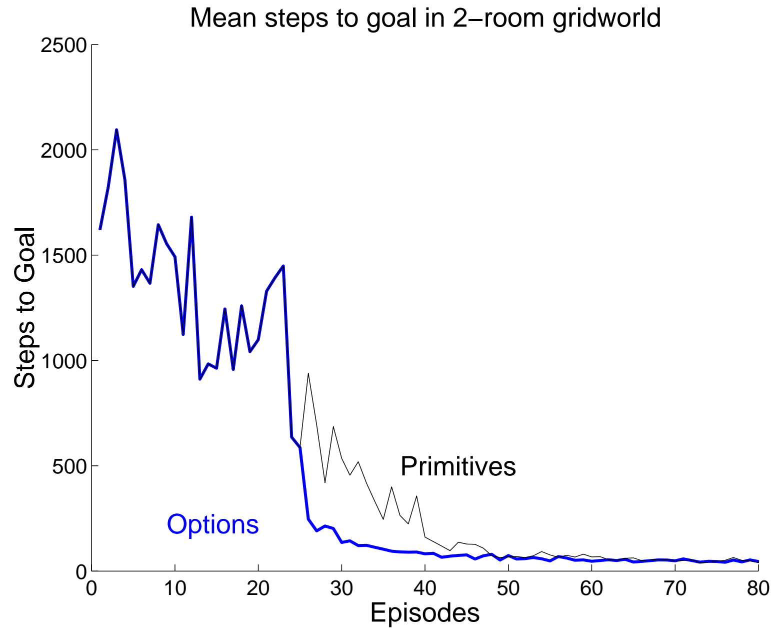


Figure 4.3. Average steps to the goal with and without automatic subgoal detection in the two-room gridworld. The results are averaged over 30 runs.

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 θ. 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.

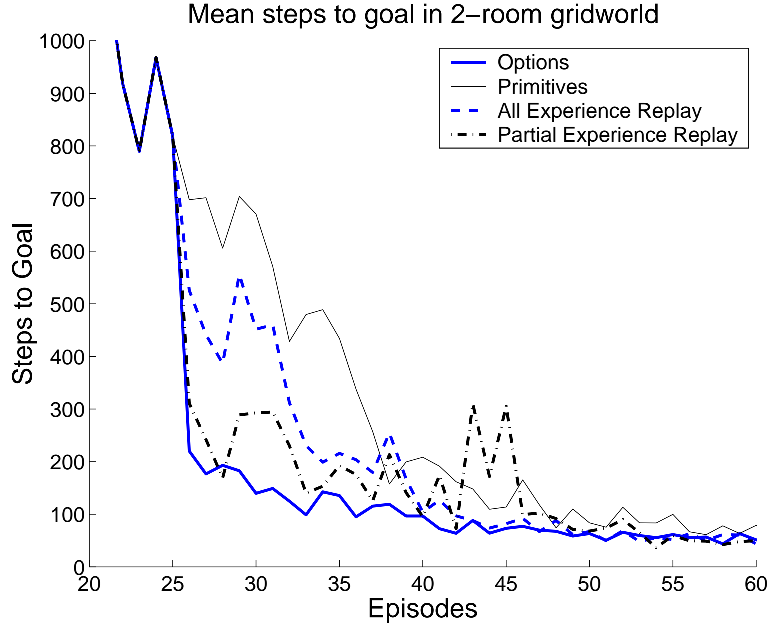


Figure 4.4. Results of using experience replay to examine the effect of the option's initial policy (partial-experience-replay) and to asses the time spent in experience replay while creating options (all-experience-replay). The results are averaged over 30 runs.

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

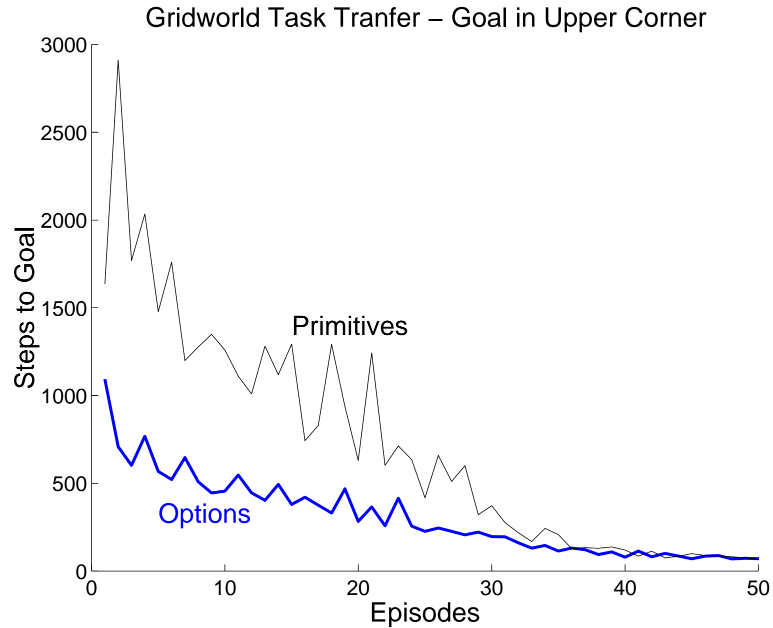


Figure 4.5. Learning curves in the new two-room task comparing the use of the previously learned options to primitive actions. The goal has been moved to the upper right corner and the results are averaged over 30 runs.

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

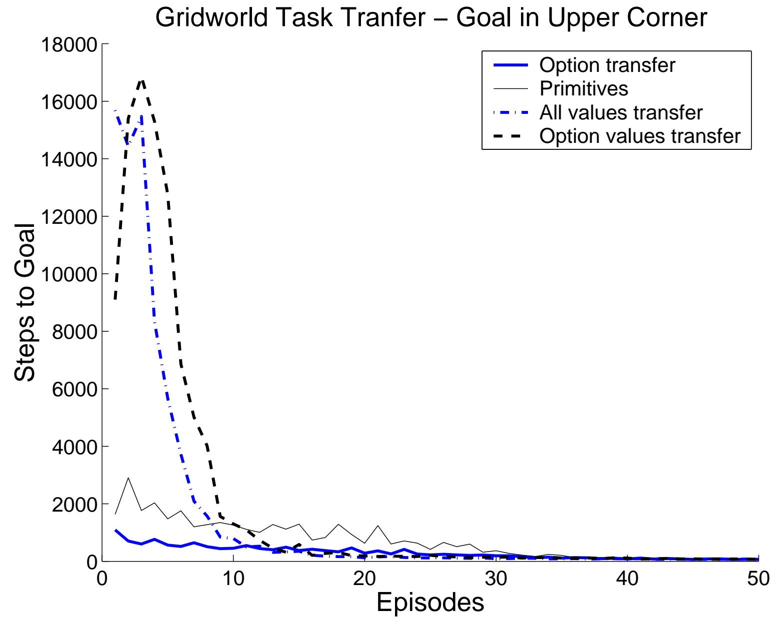


Figure 4.6. Comparison of knowledge transfer using the value function to using the newly discovered options. The goal is in the upper right corner and the results are averaged over 30 runs.

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

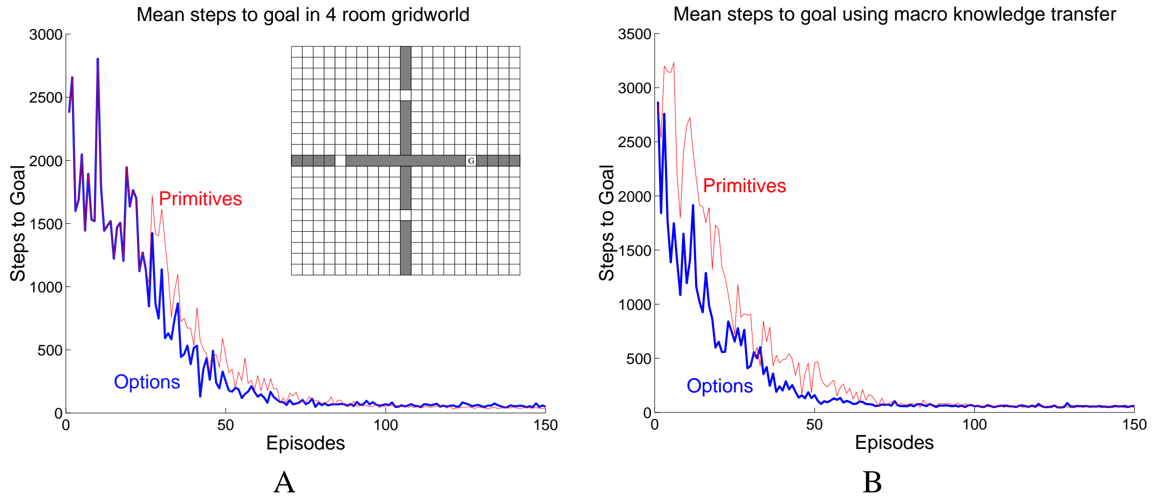


Figure 4.7. A: Comparison of automatic subgoal detection to primitive actions only in the four-room gridworld. B: Performance of the learned options on task-transfer in the four-room gridworld.

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

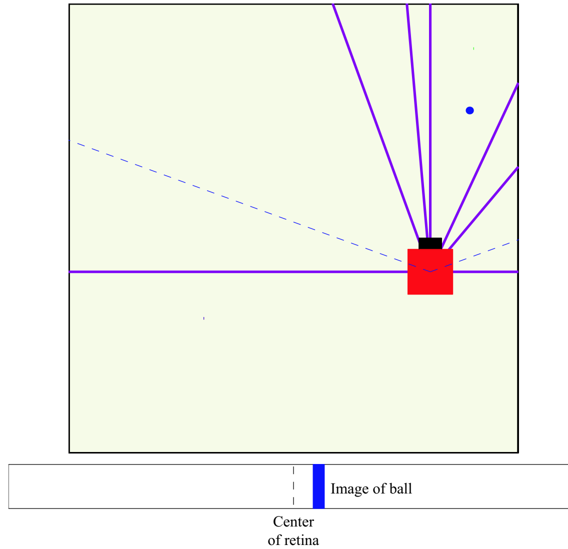


Figure 4.8. Simulated Pioneer mobile robot. The lines emanating from the robot represent the sonar beams and the two dashed lines show the vision cone. The rectangle on the bottom shows the ball's appearance on the retina.

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, γ, was 0.995, and α 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).

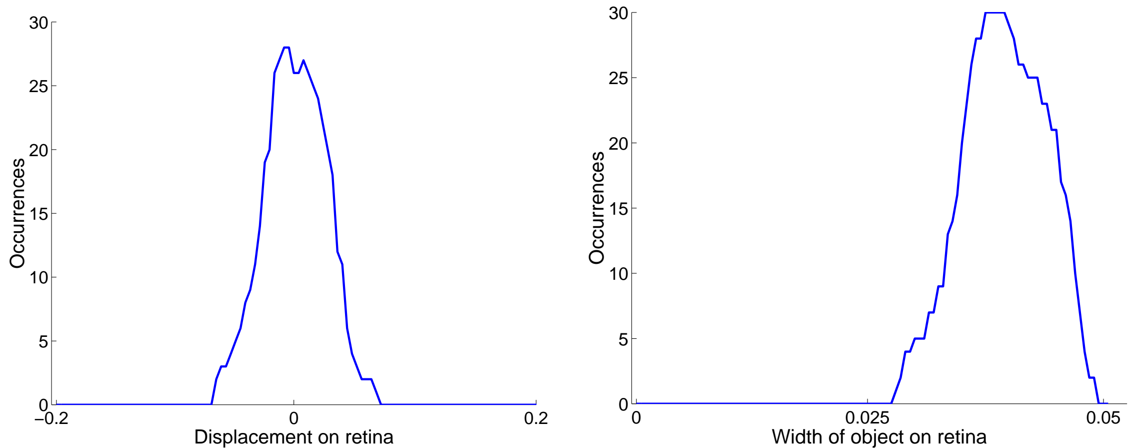


Figure 4.9. Frequencies of each sensory variable's occurrence in a subgoal summed over 30 runs. To achieve the subgoal, the robot must use both displacement and width of the ball's appearance on the retina.

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

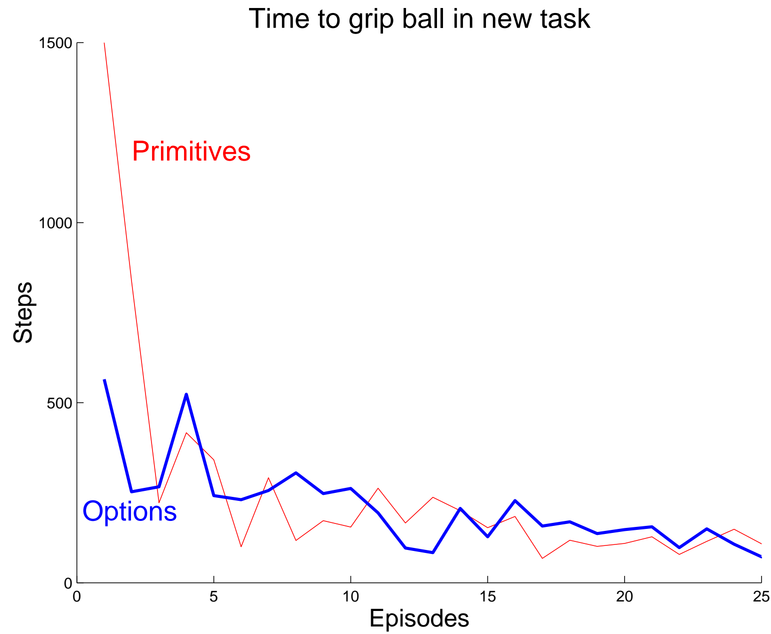


Figure 4.10. Results of using previously discovered options as compared to learning with primitive actions on a more difficult problem for the robot. These results are averaged over 30 runs.

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

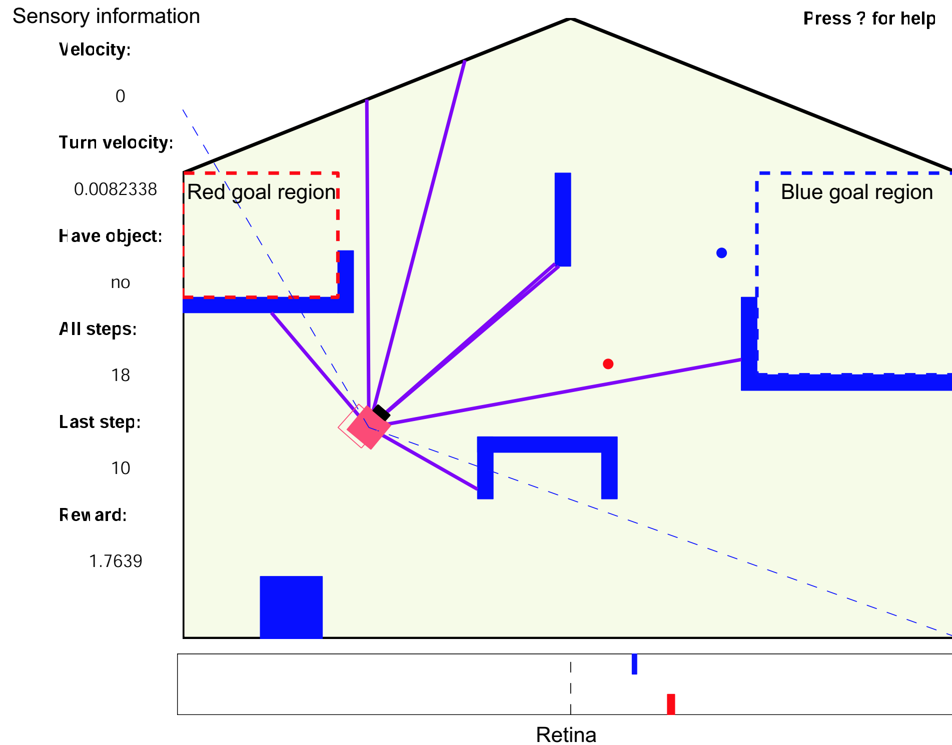


Figure 5.1. The view that the human controllers see while tele-operating the simulated mobile robot. The task is to move the blue ball to the blue goal region and the red ball to the red goal region. Sensor readings which are not visually apparent to the human from the overhead view of the robot are shown on the left hand side.

where the robot is within the room, this task is not solvable. The overhead view given in the simulator provides information of this nature to the human tele-operators.

Three types of actions are available to the robot: linear movements, rotations, and a drop action. Both the linear movements and the rotations were implemented by PD controllers that executed until the specified setpoint was achieved. The robot had a choice of three setpoints for the linear controller: 0.5, 1.0, and -1.0 units. These setpoints allowed the robot to move forward up to one unit or backward by one unit. The linear movement controller estimated the distance for achieving the setpoint using the robot's estimated coordinates. The noise in the sensing meant that the robot could undershoot or overshoot the target distance. In either of these cases, after the PD controller had terminated, the robot would have residual linear velocity that it would need to account for. For turning, the robot

had three left and three right rotation actions (5.625°, 11.25°, and 22.5°) which were also implemented by a PD controller. This controller used the estimated heading of the robot to determine if the setpoint had been achieved. The last action available to the robot was to lower and open the gripper to release any objects held inside. This was implemented as an option which was only available if an object was being held. The robot did not need a corresponding grip action as it retained the grip reflex that it used in previous experiments: if an object crossed the gripper threshold and tripped the gripper sensor, the robot immediately closed and raised the gripper.

For the experiments in this chapter, we collected data from humans tele-operating the robot. Each human operator was given a verbal description of the task before starting to control the robot. The human operators had the same set of actions available to them as described above and each action was activated by a key press. Visually, the human operators had an overhead view of the robot and its environment, shown in Figure 5.1. Sensory information that was not available from this viewpoint, such as the robot's linear and rotational velocities, was displayed as text on the side of the display. Each key press activated either a PD controller or the drop action, and the action was executed until completion. The robot on the display was updated accordingly throughout the execution of each action. Once the action terminated, the human operator was able to select a new action. An episode ended if the human operator solved the task or if the robot collided with a wall or obstacle. The human operator was provided with appropriate visual feedback in either case via a message on the display.

We collected the complete experience (sensory readings, actions, and simulator states) for each episode that the human operator controlled the robot. This allowed us to completely replay each episode of experience. We collected data from nine human operators: six computer science graduate students, one professor in computer science, and two insurance agents. This group contained two women and seven men. We collected a total of 79

Subject	1	2	3	4	5	6	7	8	9	Total
Successful Episodes	6	4	3	18	4	4	7	3	3	52
Unsuccessful Episodes	0	1	1	8	2	3	6	3	3	27
Total Episodes	6	5	4	26	6	7	13	6	6	79
Percent Success	100	80	75	69	67	57	54	50	50	66

Table 5.1. Summary of the human data collected on the tele-operated robot task.

episodes; the median number of episodes contributed by each person was six. The data for each subject are summarized in Table 5.1.

5.2 Experimental Results

Examination of the task yields two potentially useful types of subgoals: 1) foveate to and approach an object (ball) and 2) navigate to a goal region without hitting any obstacles or walls. Other types of subgoals based on other subsets of the robot's sensory information may be useful. However, these two subgoals are the most straightforward.

Using the human data, we performed two experiments to verify the hypothesis that these subgoals were discoverable by the method we introduced in Chapter 4. The first experiment created subgoals in the robot's visual observation space, and the second experiment created subgoals in the coordinate space of the robot. Both experiments used the human data to create the bags for the diverse density searches. Once the bags had been created, we used the method described in Chapter 4 to discover the subgoals. However, in Chapter 4, the experience was available to the agent in an incremental manner. To simulate the same effect here, we randomly reordered the bags before incrementally passing them to the discovery algorithm. This made it unlikely that any single person's data would dominate the evidence available at any particular moment. Although we could have run the subgoal discovery algorithm in batch mode using all of the data, the human data was still noisy and the incremental mode we developed in Chapter 4 was designed to deal with this noise. For both experiments, bags were labeled positive if the human operating the robot successfully

moved both balls to their respective goal areas, and negative if anything else occurred, such as running into a wall or obstacle.

5.2.1 Visual Subgoals

In the first experiment, we allowed the robot to create subgoals in its visual space. These subgoals were expressed by the width and displacement of an object on the robot's one-dimensional retina. This is consistent with the experiments using the simulated robot that we presented in Chapter 4 and allows us to compare results from the two experiments. We hypothesize that the methods used to detect and create the subgoal options will create very similar subgoals using the human tele-operation data as it did when using the behavior generated from the RL agent as described in Chapter 4. In the experiments using the RL data, the subgoal created was to foveate on the object and to be close to having the object in the gripper. We expected a similar "approach the object" option to be created from the human tele-operation data.

In this environment, the robot was able to distinguish between the two different colors of the balls in the room. In the experiments described in Chapter 4, there was only one color of ball in the room. In this experiment, we created two separate sets of bags, one for the blue balls and one for the red balls. The first reason that we created separate sets of bags for the two colors of balls was that it allowed us to verify our hypothesis of creating the "approach-the-object" subgoal twice, once per ball color. The second reason is that, although we expected the method to create the same "approach-the-object" concept for both colors of balls, allowing it to distinguish between the colors gave the robot the ability to create separate policies for each ball which could be useful if each ball had different dynamical properties.

We randomized the order in which both the blue and the red sets of bags were presented to the subgoal creation method and ran the method on each of the 30 different orderings. The parameters of the subgoal discovery algorithm were $\sigma_+^2 = 1.0$, $\sigma_-^2 = 0.5$, $\lambda = 0.9$, and

$\theta = 8.0$. We chose the widths of the Gaussian probability distributions to highlight the positive bags over the negative bags but to still include the negative data. This allowed the data to contain some noise in the negative bags where the object was successfully reached but the robot ran into a wall. The threshold, θ, and the decay, λ, were chosen as in Chapter 4, where we wanted the concept to appear early in the data and to persist. We again used an incremental clustering algorithm to store the potential concepts and the frequencies of how often each concept was found. New concepts were defined to be in an existing concept cluster if the Euclidean distance was one or less. This distance was calculated using the two sensory features of the width and displacement of the object on the robot's retina. These two features were scaled to have approximately equal ranges.

The subgoals that the agent created for both the red and the blue balls over all 30 runs are shown in Figure 5.2. Each subgoal is expressed as a conjunction of the two features. The figure shows that the robot created subgoals where the ball is in the center of its retina and the ball spans almost its maximum width on the retina, which means that the robot is close to picking up the ball. This is a useful subgoal for any task involving the robot moving the ball, as we demonstrated in Chapter 4. The difference between the subgoals for the two colors of balls seems to come from the human subjects' tendencies to move the blue ball to the goal before the red ball. This meant that the humans ran the robot into walls more frequently with a blue ball in the gripper which affected the results by moving the maximal concept to having the ball span a smaller width on the robot's retina, or to be slightly farther away from the blue ball.

These results are important for several reasons. The first is that by generating subgoals from the human data that correlate with the subgoals generated from the RL data, we demonstrate that our subgoal creation method is more generally applicable beyond RL. The method is able to generate useful subgoals from experience other than the online behavior of an RL agent, which means that subgoals can be discovered in tasks that an RL agent may not necessarily be able to solve. Although we used human tele-operation data

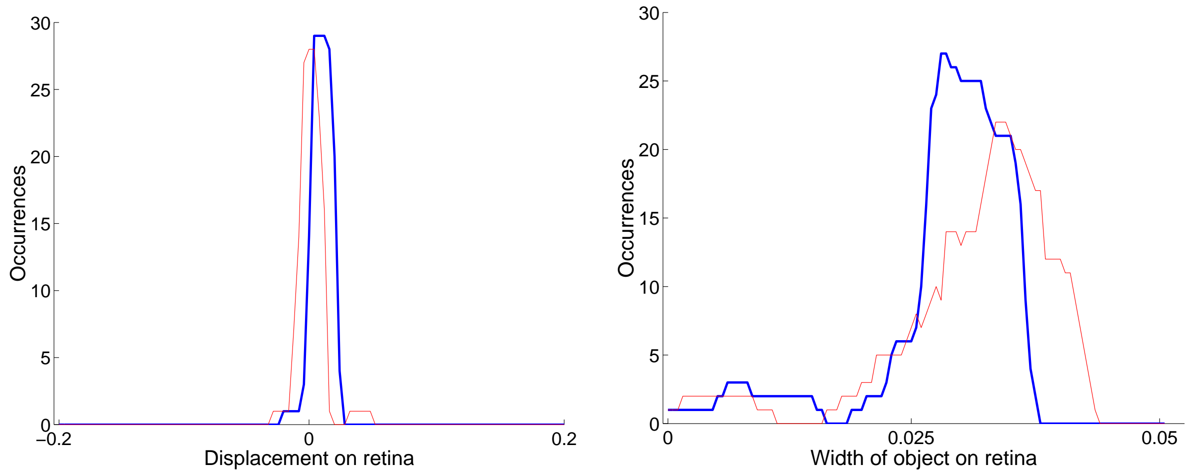


Figure 5.2. Frequencies of each sensory variable's occurrence in a subgoal generated from the human data summed over 30 runs. To achieve the subgoal, the robot must use both displacement and width of the object's appearance on the retina. The results for the blue balls are shown with the thicker lines and the results for the red balls are shown with the thin lines.

for these results, the behavior of any purposeful agent that can successfully solve the task could be used to detect and create the subgoal options.

5.2.2 Navigational Subgoals

Once the robot knows how to pick up objects in the environment, it needs to know how to navigate to the red and blue goal locations without running into any obstacles. These two locations can be discovered as subgoals using the diverse density approach described in Chapter 4. These subgoals are expressed using the robot's estimated position in the room. The other sensory readings available to the robot are not included in these subgoals.

The bags for this experiment were created using the estimated positions of the robot, which means that each bag contained each of the coordinates that the robot visited on a single trajectory. As in the previous experiment, bags were labeled as positive if both balls were successfully brought to their respective goal regions and negative otherwise. Because there were two potentially useful subgoal locations (one for each goal region), we used the disjunctive-point concept class described by Maron (Maron, 1998). This is a simple

extension of the single-point concept class described in Chapter 4 where the probability of a particular instance B_{ij} being in the concept $(c_1 \bigvee c_2)$ is the maximum of the probability of the instance being in the individual single-point concepts c_1 and c_2. This generalizes to any number of disjunctive single points. For this experiment, we used 2 disjuncts (one for each goal location).

The ordering of the bags was randomized before being incrementally presented to the discovery algorithm. The parameters for the diverse density and the discovery algorithms were $\sigma_+^2 = 1.0$, $\sigma_-^2 = 1.0$, $\lambda = 0.9$, and $\theta = 8.0$. In this case, we chose to weight the positive and negative bags equally. We chose the standard deviation, σ^2, of the probability distribution used by diverse density to be 1.0 because the robot's size is one unit square. The incremental clustering used a Euclidean distance of one to determine if two concepts were the same, and the static filter constrained the two subgoals to be a Euclidean distance of at least five apart.

Figure 5.3 shows the location of the subgoals in the environment for all 30 runs. Each subgoal location is shown as an asterisk. As the figure shows, the agent discovered two main subgoals, each close to the goal location for each color of ball. The subgoals are far enough away from the obstacles to make navigation to each subgoal safe for the robot. The subgoals near the red goal location are clustered into two separate regions. Examination of the human tele-operation data shows two distinct behaviors, each of which accounts for one of the two red subgoal regions. The first behavior was that, after running into a wall, the human operators learned to keep the robot as far as possible from walls and sharp corners. The second characteristic of the human tele-operation data was that many of the human operators chose to drive the robot above the middle obstacle when navigating from one goal region to another. Although the human operators did not choose this route on each episode, none of these trajectories were unsuccessful which means that the diverse density values for the common areas of these trajectories were high. Although the latter set of

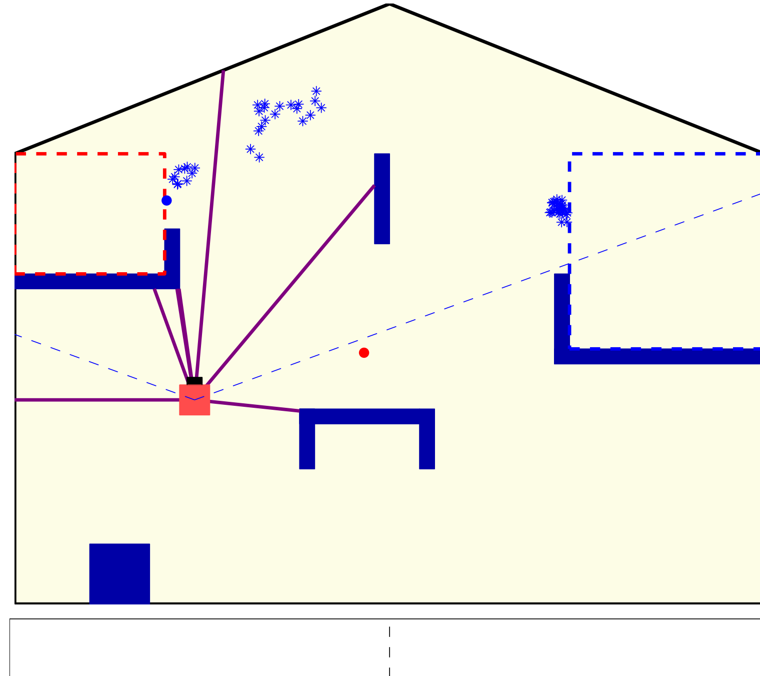


Figure 5.3. Navigational subgoals discovered by the agent over 30 runs. Each subgoal location is shown as an asterisk.

subgoals is not immediately adjacent to the red goal region, it should be helpful to a robot learning to navigate in this environment.

Although we do not have corresponding navigational subgoals to compare against that were created from RL data, these subgoals correspond with our own intuition of what useful subgoals should be in this task. If the input set of the options create to reach the subgoals includes the entire room, it will significantly affect the exploration of the robot. These options will cause the robot to visit the goal regions with a higher probability than other regions of the room. This effect should accelerate the learning of an agent on this and related tasks.

5.3 Conclusions

We have demonstrated that the data from humans tele-operating a simulated robot in a complex task can be used to generate useful subgoals. These subgoals are detected and

created using the method introduced in Chapter 4. These experiments demonstrate that the method can generate useful abstractions in tasks that may be difficult for an autonomous machine learning agent to solve but where an outside agent can provide example trajectories. These trajectories do not need to be optimal but can be generated from exploratory but purposeful behavior.

The fact that the subgoal creation method can generate useful subgoals from the behavior of multiple types of purposeful agents increases the range of applications where the method can be used. Situations ranging from small simulated tasks to robots operating in space may be able to benefit from this method of automatically creating subgoals.

CHAPTER 6

FINDING USEFUL SEQUENCES

The second type of temporal abstraction that we focus on in this dissertation is an action sequence. An action sequence is an open-loop policy that encapsulates a sequence of actions into one new action. An open-loop policy chooses its actions at each step independent of any feedback from the environment. This is in contrast to closed-loop policies, used in Chapter 4 to achieve the subgoals, that choose the actions at each step in response to feedback from the environment. The action sequences are similar to the idea of chunks in SOAR where the agent creates new high-level actions from useful sequences of lower-level actions (Laird et al., 1986). In this chapter, we first present a method for automatically discovering useful sequences from the online behavior of an RL agent and then illustrate the advantages and drawbacks of the method on several different gridworlds. Chapter 7 focuses on the use of this method for sequence discovery on the real-world problem of instruction scheduling.

The method that we use to discover useful action sequences is based on searching for sequences that occur frequently on successful action trajectories. This idea is related to the bottleneck ideas described in Chapter 4, where the agent looks for observations that occurred across multiple successful trajectories but not in unsuccessful ones. Unlike the diverse density approach used to detect the subgoals, the method introduced in this chapter does not use negative evidence in discovering the sequences. It may be possible to adapt the diverse density methods to search for sequences but it would be a more inefficient approach than the one presented here.

The method presented in this chapter can create two forms of open-loop action sequences. In the first form, once the agent has selected the sequence as its current action, it must execute each of the lower-level actions in the sequence before it can choose a new high-level action. The second type of sequence that the method can create allows the agent to conditionally stop a sequence while it is executing if the agent visits an unexpected state or experiences an unexpected observation. We denote this latter type of sequence as a conditionally terminating sequence. We refer to the first type as an action sequence.

The ability to form new actions out of sequences of actions has several advantages to an agent. Both types of sequences give an agent the immediate ability to search more deeply in a search tree by shortening the path to a solution. For example, instead of having to search through three levels of `up` actions, a gridworld agent can achieve the same effect in one level by using one `up3` sequence. This advantage is also present for the subgoal options that we introduced in Chapter 4. A related advantage of using sequences as action primitives is that they enable an agent to quickly build a hierarchy of actions. At the higher-level, the actions are sequences of primitive actions and other sequences. The lowest level actions are primitive actions. Other researchers (e.g., Dietterich, 1998; Kaelbling, 1993; Dayan and Hinton, 1993) have shown that action hierarchies can facilitate both learning and knowledge transfer. Although subgoal options can be used to build multiple levels of action hierarchies, it is much more difficult than with sequences.

Another advantage of using sequences is that, even though open-loop policies are usually considered to be a disadvantage because of their inability to respond to stimuli in the environment, they require less computational overhead than closed-loop policies because of the lack of sensing actions. This can allow an agent to explore a new environment more rapidly. By trading off some of the computational savings of open-loop sequences for effort spent in sensing actions, it is also possible to create conditionally terminating sequences of actions. This type of sequence can provide some of the same advantages as the action sequences as well as an ability to terminate if the environment changes dramatically. Neither

type of sequence can modify its specific sequence of actions. We will explore both types of sequences in this chapter. Another approach to the tradeoff between open-loop policies and sensing actions can be found in Hansen et al. (1997).

Action sequences can also be disadvantageous to an agent in some situations. For example, even though moving upward three times in a row requires little computational effort to execute, a gridworld agent might be penalized for hitting a wall two out of the three moves because it did not check the sensory readings before executing the last two up actions. Although an RL agent can learn not to choose the `up3` action in such states, it would be better if the agent learned a sequence such as `go-up-until-wall`. The conditionally terminating sequences are closer to this ideal than the action sequences. Alternatively, if the sequences can only be chosen in a certain set of states, the agent might not have the opportunity to choose the `up3` sequence when it is near a wall. Although this could solve the problem, specific knowledge of each task where the option might be used is necessary. The optimal set of states in which each option should be applied is task dependent.

6.1 Sequence Discovery

Detecting useful sequences from various types of data is not a new issue in computer science. One well known algorithm, Longest Common Subsequence (LCS) (Cormen et al., 1990), is an efficient algorithm for finding the longest common sub-sequence across multiple sequences. However, this algorithm does not satisfy the requirements of the problem that we are interested in for two main reasons. The first reason is that data used to create the sequences are typically generated from the online behavior of a learning agent. This behavior is inherently noisy because of the exploration necessary to learn good policies. Stochastic environments contribute to the noise in the data as well. This means that the useful sequences may not be common to every trajectory, as LCS requires. The second reason that LCS is not applicable is that the longest sequence may not be the one that is most useful to an agent because the longest sequence will not necessarily be the most widely ap-

plicable sequence. For example, once an optimal policy is learned, the agent could chunk the entire sequence of actions from the start state to the goal state. However, this is not likely to facilitate learning on any other tasks or on the current task, if it is stochastic.

Sequence detection has also been studied in the knowledge discovery, information retrieval, and bio-informatics fields. This work is discussed in more detail in Chapter 3. We focus on Zaki's (1998, 2001) work on generalized sequence detection in this chapter because it is most closely related to our research. His methods are aimed at finding predictive patterns in large collections of data such as those generated by tracking each customer's purchases at a grocery store. He looks for rules such as: "85% of people who bought baby food also bought diapers within a week." This approach could be adapted to finding action sequences by searching for patterns such as: "after an agent goes up twice, it chooses to go right next in 75% of the cases."

Although Zaki's algorithm runs efficiently in average case scenarios, it has two disadvantages. The first is that our sequence tasks do not fall into his average case scenario but rather into his worst case scenario, mainly because we have only one item per time step which negates the efficiency of his database structure. Because of this, the sequences take exponential time to detect. Second, we are more interested in sequences where the actions are consecutive. Zaki's algorithm can detect sequences that are not contiguous in time, such as "go up now and left 3 steps later", but it is not clear how an agent would use such a sequence. The requirement that the actions in a sequence occur consecutively in time with no gaps significantly reduces the search space for any sequence detection algorithm.

Zaki does provide terminology and theory that we use to construct our algorithm to run efficiently. We look for sequences with *support* at least $p \in [0,1]$, which means the sequence occurs in at least a $100p\%$ of the trajectories in the database. Zaki proves that for any sequence ψ of length n with support at least p, all subsequences of ψ of length less than n must also have support at least p. A sequence $\psi' = \{a_1, a_2, \ldots, a_k\}$ is a subsequence of sequence $\psi = \{b_1, b_2, \ldots, b_n\}$ if and only if $a_1 = b_i, a_2 = b_{i+1}, \ldots, a_k = b_{i+k}$ for some

$b_i, \ldots, b_{i+k}$ where $i+k \leq n$. This fact about subsequences having at least the same support as the larger sequence is one reason that both of our algorithms can run efficiently.

6.1.1 Consec: The Sequence Detection Algorithm

The sequence detection algorithm that we introduce is summarized in pseudocode in Tables 6.1 and 6.2. We first discuss `consec`, our sequence detection algorithm, and then we discuss the method to create options from the sequences.

Building on the knowledge that all subsequences of a supported sequence must have at least the same support level, we can efficiently construct the complete set of frequent sequences using a classical technique known as recursive doubling. In this context, recursive doubling means that the sequences can be grown by starting with a set that is easy to calculate (e.g., the supported sequences of length one) and doubling the length of the sequences at each step. This means that the sequences are created for lengths $1, 2, 4, 8, \ldots$ until no more supported sequences exist. The missing sequences, i.e. those of lengths $3, 5, 7, \ldots$, are detected using a different method that is still efficient. By using this approach, `consec` cuts down the number of passes required to build a sequence of length n from $O(n)$ to $O(\log n)$. For applications in which `consec` must run frequently, or where there are long or numerous sequences, this is a considerable improvement in running time.

`Consec` starts by finding the sequences of length one with the specified level of support. This is accomplished efficiently in one pass through the database. The total number of primitive actions is known beforehand and the algorithm checks each trajectory for the existence of each possible action and increments a counter if the action exists within the trajectory. This count gives the support for each action directly.

After finding the frequent sequences of length one, `consec` moves to its recursive doubling phase (this is denoted "find sequences of length ≥ 2" in Table 6.2). In this phase, the recursion ends if the list of sequences to double is empty. If this is not the case, then the algorithm initializes both a failure and a success list to be the empty list. Next, it loops

```
consec
    input: support p
           minimum length l
    output: list of supported sequences all-seqs

    1. seqs1 ← find all sequences of length 1(p)
    2. long-seqs ← find all sequences of length ≥ 2 (p, seqs1, seqs1)
    3. all-seqs ← long-seqs ∪ seqs1
    4. remove any all-seqs with length less than l
    5. return pruned all-seqs

find all sequences of length 1
    input: support p
    output: supported primitive actions

    1. initialize counts for each action to 0
    2. for each trajectory t in the database
    3.     for each action a
    4.         if action a ∈ t
    5.             counts(a) ← counts(a) + 1
    6. minimum-trajectories ← p × number of trajectories in the database
    7. return all actions a with counts(a) ≥ minimum-trajectories
```

Table 6.1. Pseudocode for `consec`, the sequence detection algorithm, and the method to find supported primitive actions.

find all sequences of length $\geq$ 2

input: support *p*

supported primitive actions *seqs1*

list of sequences currently being doubled *current*

output: all supported sequences

1. if current is $\emptyset$, then return
2. Initialize fail and success to $\emptyset$
3. for each sequence s1 $\in$ current
4. for each sequence s2 $\in$ current
5. new-seq $\leftarrow$ s1 + s2
6. if new-seq has support p
7. success $\leftarrow$ success $\cup$ new-seq
8. else
9. fail $\leftarrow$ fail $\cup$ s1
10. doubled $\leftarrow$ find all sequences of length $\geq$ 2(p, seqs1, success)
11. slow $\leftarrow$ grow slowly (p, seqs1, fail, 2 $\times$ length of sequences $\in$ current)
12. if fail is $\emptyset$
13. slow2 $\leftarrow$ grow slowly (p, seqs1, current, 2 $\times$ length of sequences $\in$ current)
14. return $\leftarrow$ doubled $\cup$ success $\cup$ slow $\cup$ slow2

grow slowly

input: support *p*

supported primitive actions *seqs1*

list of sequences currently being grown *current*

maximum length that the sequences can achieve

output: all supported sequences of less than the specified length

1. if current = $\emptyset$ or length of sequences $\in$ current $\geq$ maximum length, then return
2. success $\leftarrow$ $\emptyset$
3. for each sequence seq $\in$ current
4. for each each sequence $s1$ $\in$ seqs1
5. new $\leftarrow$ seq + s1
6. if new has support p
7. success $\leftarrow$ success $\cup$ new
8. slow $\leftarrow$ grow slowly(p, seqs1, success, maximum length)
9. return $\leftarrow$ success $\cup$ seqs $\cup$ slow

Table 6.2. Pseudocode for the methods used by `consec` to find all sequences of length two or more using recursive doubling.

through the current list of supported sequences and combines them. This is the doubling phase of the recursive doubling approach. If the new sequence has enough support, it is added to the success list. Otherwise, the original sequence is added to the failure list. When each of the current sequences has been either successfully doubled or has failed to combine with another sequence, the algorithm calls itself recursively on the success list. Finally, the failure list is examined to see if any longer (but less than double length) sequences exist with enough support. A special case arises when all sequences are successfully doubled. In this case, the list of original sequences is also passed to the method that grows the sequences one item at a time. If this case is ignored, the algorithm can miss some of the shorter sequences. The combination of the sequences that were successfully doubled and those which were successfully grown without doubling is returned to `consec`.

The mechanism for growing the sequences that fail to double is also implemented in a recursive manner. This method is denoted "grow slowly" in Table 6.2. This method takes an additional parameter specifying the maximum length that its sequences can reach. This length is known in advance because these sequences have already failed to double. The method starts by ensuring that the length of the sequences in the current set to grow is still acceptable. If this condition holds, then each sequence in the current set is grown by adding each of the set of length-one sequences to it. If the new sequence has an acceptable level of support, it is saved. The method recurses on this list of new sequences. All successfully expanded sequences are returned at the completion of the method.

Once the list of supported sequences has been created, `consec` prunes the list according to user specification. Sequences of length one, and possibly other short sequences, are not of interest to an agent trying to add new actions to its set because they coincide with primitive actions. Instead, we include a parameter that specifies the minimum length of any sequence that `consec` can return. `Consec` removes any sequence shorter than the minimum length before returning the list of supported sequences.

1. Initialize full trajectory database to $\emptyset$
2. Initialize running averages ρ_s to 0
3. For each episode
4. Interact with environment (Potentially learn using RL)
5. Add observed full trajectory to database
6. Use `consec` to search for sequences
7. For each sequence s found
8. Update the running average by $\rho_s \leftarrow \rho_s + 1$
9. If ρ_s is above threshold θ
10. If s passes static filter
11. If s is an action sequence
11. Initialize I by examining trajectory database
12. Set β to be 1 when the sequence is terminated and 0 otherwise
13. else
14. Initialize I by examining the sensory component of s
15. Set β to be 1 when the sequence terminates or if the sensation does not match s and 0 otherwise
16. Set policy π to be the sequence s
17. Create a new option $o =< I, \pi, \beta >$
18. Decay all running averages by $\rho_s \leftarrow \lambda \rho_s$

Table 6.3. Pseudocode for the sequence-creation algorithm.

6.1.2 Sequence Detection and Creation

We use `consec` to detect the frequent sequences of actions and observations. These sequences are used to create new options and the method for doing so is summarized in Table 6.3. This method is very similar to the subgoal detection and creation algorithm that we presented in Chapter 4.

The method starts by observing the behavior of a purposeful agent. The agent saves its full experience during each episode. Each episode is marked as successful or unsuccessful, according to the current definition of success. "Success" can be defined on a per-task basis but, in most cases, it means that the agent accomplished its goal within a specified number of steps and that it did not visit any failure states.

When enough data has been collected, the method uses `consec` to search for the frequent sequences within the successful episodes. If the trajectories are being generated by

the online behavior of a learning agent, then the initial experience may be very noisy as the agent explores its environment. Stochastic environments will also contribute to the noise. If the trajectories are being generated by other means, such as heuristic search, there may be less variation across trajectories. Assuming that there will be some noise due to exploration or stochasticity in the environment, the method maintains a running average of how often each sequence has been detected by `consec`. This average is updated as described by Equation 4.8. As before, we use a threshold, θ, to determine if a sequence has been found frequently enough to consider it as a possible new action. This threshold can be chosen to account for the amount of noise in the data by setting it low (little noise) or high (very noisy). The running average and threshold help to prune the sequences to those which appear frequently in the supported list. The idea of searching for "early and persistent" bottlenecks also applies here. The early and persistent sequences are more likely to reflect features of the environment than the later sequences which constitute an optimal policy for the task. Once the running average for a sequence rises above the threshold, then that sequence is passed to a static filter.

The static filter (Iba, 1989) ensures that new sequences are not created if they are too similar to currently existing actions. It can also be used to impose any task-specific constraints on new actions. The subgoal discovery method used a static filter for similar reasons. To ensure that the sequences are not too similar to one another, we need to define a distance metric between sequences. We were able to do this in Chapter 4 for subgoals, but it is much more difficult to define a distance metric for a sequence of actions than it is for subgoals in sensory space. With many types of subgoals, it is possible to say that a location in sensory space, e.g. (x, y), is a specified distance away from other locations such as (x', y') using standard measurements such as the Euclidean distance. For sequences, appropriate distance metrics are more task dependent and rely on knowledge of the consequences of each action. For example, in a gridworld, the action sequence `right`, `up`, `right`, `up` usually has the same effect as the action sequence `right`, `right`, `up`, `up`. A distance

metric for sequences in a gridworld should calculate that these particular sequences are a distance of zero from one another. Using the Manhattan distance between the expected consequences of successfully completing each sequence would satisfy this requirement. In other environments, such as the instruction scheduling task discussed in Chapter 7, an appropriate distance metric can be even more difficult to determine. We discuss the issues for this task more fully in Chapter 7.

Once an action sequence successfully passes the static filter, it can be added to the action set of the agent. We continue to use the options framework to represent the temporal abstractions, in this case, both types of sequences. The policy, π, of the new option is set to be the sequence of actions. In the case of action sequences that do not conditionally terminate, the termination condition, β, is set to one at the completion of the sequence, and zero otherwise. In the conditionally terminating case, β may depend on the results of a sensing action. For example, if a gridworld agent finds the sequence `up`, `wall-check`, `up`, `wall-check`, then β will be set to one in any state where there is a wall or at the completion of the sequence, and zero otherwise.

The input set, I, also depends on whether the sequence is conditionally terminating or not. If it is not, I is initialized using a heuristic similar to that used to initialize I for the subgoal options. More specifically, each of the saved successful trajectories is searched for occurrences of the action sequence. Every state in which the sequence began to execute is added to the input set. This means that the input set is the union of all the states in which the sequence had begun from all of the successful saved trajectories.

For conditionally terminating sequences, `consec` returns not just a sequence of actions but a sequence of observations associated with those actions. The input set is initialized using the sequence of observations by allowing the new option to be chosen in any state where the agent's current observation matches the first observation in the sequence. Once a conditionally terminating sequence has been selected by the agent, the agent continues to check its observations against the observation sequence at each step. If the two observations

do not match, the option is terminated. This approach leads to an ability to generalize to unseen situations assuming that the agent's set of sensory readings remains consistent across tasks.

6.2 Experimental results

In this chapter, we use gridworlds to illustrate different aspects of the automatic sequence creation algorithm. The following chapter presents results for a real-world instruction scheduling task. We first concentrate on discovering action sequences and explore advantages and disadvantages to their use. Next, we explore the use of conditionally terminating sequences.

6.2.1 Experiments with Action Sequences

Action sequences can facilitate learning both within and across tasks. The first task that we present to the sequence discovery algorithm is a 20x20 stochastic gridworld with no obstacles. The agent had four primitive actions available to it: `up`, `down`, `right`, and `left`. These actions were stochastic in their effects where the action moved the agent in the indicated direction with probability p and in a random direction with probability $(1-p)$. For the first experiment, the actions succeeded with probability 0.9. The agent's task was to navigate from the lower left corner of the gridworld to the upper right corner. It received a reward of $+1$ for reaching the goal and 0 otherwise. The discount rate, γ, was set to 0.9, so that that the agent tried to reach the goal in as few steps as possible. The performance for each of the experiments in this chapter was measured by the number of primitive steps that the agent took to reach the goal state from the start state. This is averaged over 30 different runs and compared across learning algorithms.

We compared the performance of an RL agent learning using only primitive actions to that of an agent using the sequence discovery algorithm. The agent explored during learning using a 5% exploration rate. After each episode, the agent evaluated its action-value

function greedily. For this experiment, the sequence discovery algorithm created action sequences and not conditionally terminating sequences. The parameters of the sequence discovery task were support = 0.9, minimum sequence length = 4, $\theta = 9.0$, and $\lambda = 0.9$. The support level was set to a fairly high number to filter out any sequences that appeared only a few times during exploration. The running average parameters were also chosen for this reason: the sequence had to appear with a high probability and it needed to appear frequently enough for the running average to be close to its maximum. The last parameter, the minimum length of the new sequences, was chosen such that the new sequences would be significantly different from primitive actions. The agent was restricted to creating only one new sequence per run to allow us to examine how even one sequence could affect learning. The static filter accepted any proposed sequence. In addition, we defined successful trajectories to be only those in which the agent took 1000 steps or fewer to reach the goal. The sequence detection algorithm, `consec`, was not initiated until at least 10 successful trajectories had been observed. For reasons explained in Chapter 4, the learning rate was set to 0.05.

Figure 6.1 compares the performance of the RL agents learning with and without sequence discovery. The sequence discovery results are graphed using a thicker line. These results are averaged over 30 different runs, each starting with a different random seed. The two agents share the same random seed on any particular run. The agent using sequence discovery was able to learn an optimal policy more quickly than the agent without sequence discovery, which demonstrates that the creation of action sequences can be useful within a particular task. The initial episodes of both agents are the same because the sequence discovery algorithm does not create any new sequences until approximately episode 35.

We hypothesized that useful sequences in this particular gridworld would be those that moved the agent up and to the right. To verify this hypothesis, we graphed the sequences that the algorithm actually discovered in the following way. Each action in the sequence was assumed to succeed with probability one. For each sequence, the agent was assumed to

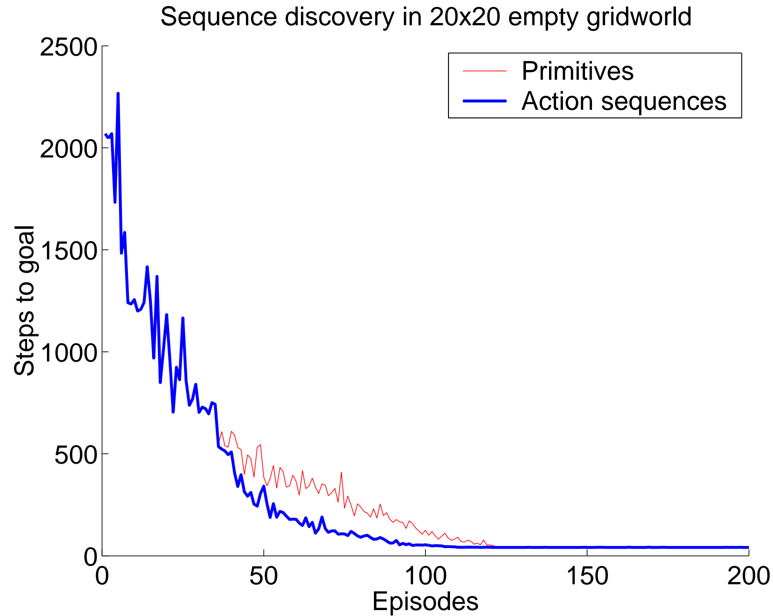


Figure 6.1. Performance comparison between an RL agent automatically discovering new action sequences and an agent learning using only primitive actions.

start in position $(0,0)$ and each action in the sequence either moved the agent `up` $(+1,0)$, `down` $(-1,0)$, `right` $(0,+1)$, or `left` $(0,-1)$ one step from its current position. The final position of the agent, at (r,c), is recorded at the completion of each sequence. We drew a line from $(0,0)$ to (r,c) to represent the effect of each action. Representing the results in this manner corresponds to our knowledge that, in a deterministic gridworld, the action sequence `up`, `right`, `up`, `right` has the same effect as the action sequence `up`, `up`, `right`, `right`. Figure 6.2 shows the sequences in this form for each of the 30 runs. The numbers next to each line correspond to the number of runs that discovered that sequence. As we hypothesized, the sequences move the agent up and to the right.

In the next experiment, we examined how the reuse of the learned action sequences affected the learning ability of the agent in similar environments. In this experiment, we used a 40x40 gridworld with no obstacles where the goal state was again placed in the upper right corner and the start state was in the lower left corner. This environment is similar to the one described above except that the new environment is larger (40x40 versus

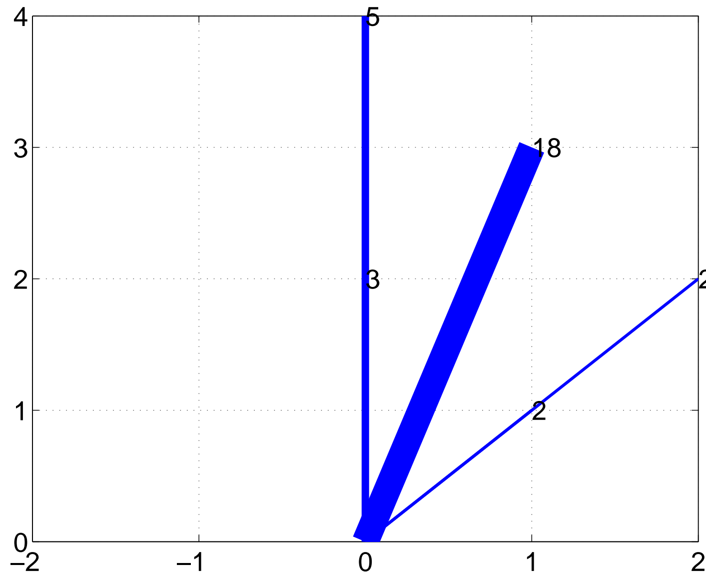


Figure 6.2. Action sequences discovered by the agent in the first gridworld task. Each sequence is shown as a line from the origin to where the sequence would move the agent assuming that all actions in the sequence succeeded. The width and the numbers next to each line represent the number of different runs in which that the sequence was discovered.

20x20) and the effect of the actions is more stochastic ($p = 0.8$ versus $p = 0.9$). Both of these attributes make this task harder for the agent than the first task. We compare the performance of an RL agent learning with only primitive actions to that of an agent learning with primitive actions and with the action sequences discovered in the first task. The agent created 30 action sequences in the original task, one sequence per run. For this task, the agent reused the sequences from the corresponding run of the previous task. The learning parameters remained the same.

Figure 6.3 shows the results of comparing the average performance of the two agents over 30 runs. The results for the agent reusing the action sequences are shown with a thick line. The agent using the action sequences was able to find the goal in fewer steps than the agent with only primitive actions even in the beginning stages of learning. This advantage continues throughout learning where the agent with the action sequences converged to an optimal policy more quickly (at about episode 425 versus 500). This experiment demon-

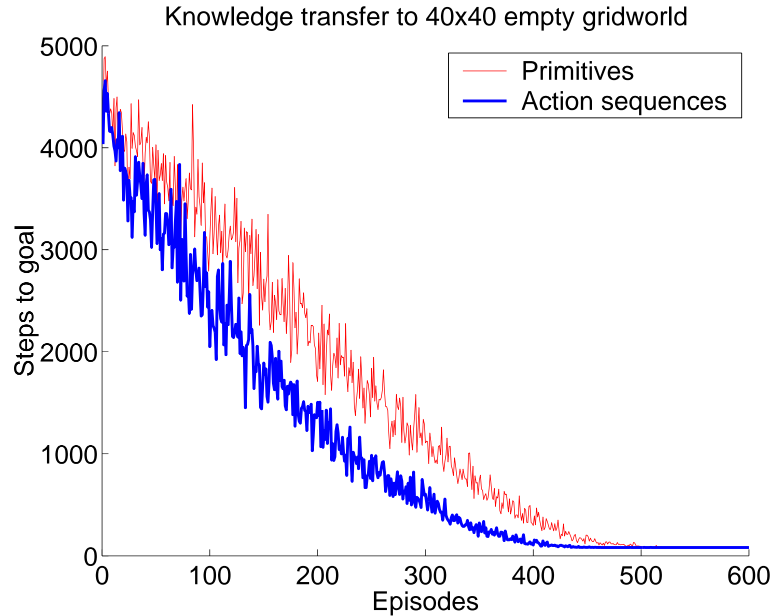


Figure 6.3. Performance comparison between the agent reusing the sequences learned in the first task on a new and larger gridworld to an agent learning using only primitive actions. The results for the agent using knowledge transfer are shown with a thick line.

strates that the action sequences were able to facilitate knowledge transfer by significantly accelerating the learning of the agent in the new task.

Any type of abstraction suffers from the fact that although it may be useful for one set of tasks, it may not be useful for other tasks and could even be detrimental to the agent. This is particularly true for action sequences because they execute without considering feedback from the environment. If the environment changes significantly between tasks, this open-loop execution can impede learning instead of facilitating it. For example, if the agent has learned sequences to move it up and to the right, as in our first experiment, and it is presented with a new task where the goal has moved to the upper left corner or a task where a wall has been added that the agent must navigate around, learning could be slowed.

We tested this hypothesis by examining the performance of a learning agent using the set of discovered sequences and comparing to the performance of an agent using primitive actions only. We examined both of the cases described above. The size of each of the

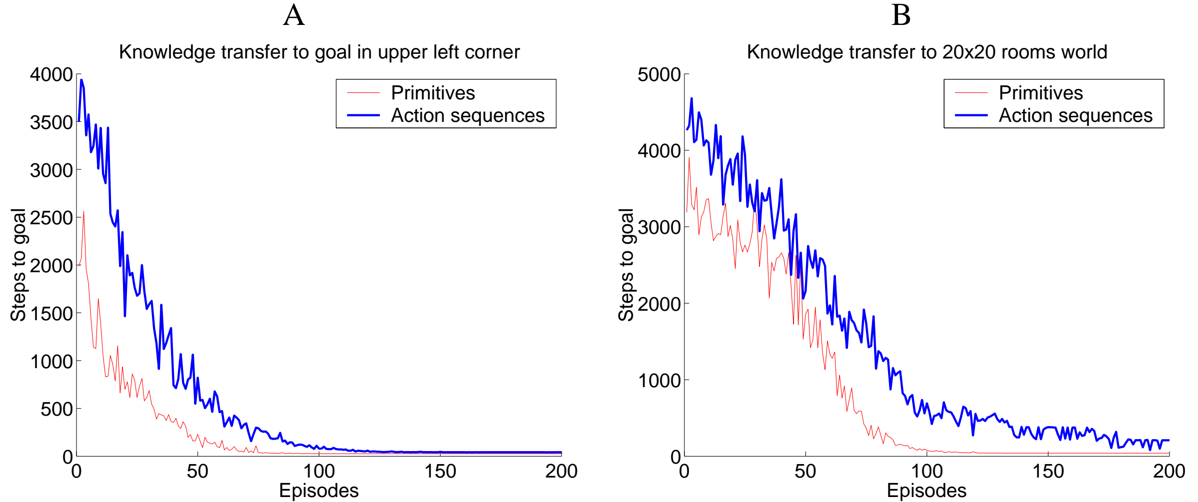


Figure 6.4. Performance comparison between the agent reusing the sequences learned in the first task on the task with the goal moved to the upper left corner (Panel A) and in the 20x20 gridworld with a wall inserted in the middle (Panel B). The knowledge transfer results are shown with a thick line.

gridworlds is the same as the first gridworld where the agent learned the sequences: 20x20. The stochasticity and learning parameters also remained the same. In the first case, the task was modified by moving the goal to the upper left corner and, in the second case, a wall was inserted into the middle of the environment with a doorway in the middle of the wall. The wall spanned the gridworld from top to bottom. Figure 6.4 shows the comparison between the two learning agents for both of these environments. In both cases, the performance of the agent reusing the action sequences is shown as a thick line. As we expected, the use of the action sequences impeded learning in both tasks. One method for addressing this issue is to use conditionally terminating sequences that can end when the environment changes so radically.

6.2.2 Experiments with Conditionally Terminating Sequences

The algorithm that we used to detect and create new action sequences can also be used to automatically discover useful conditionally terminating sequences by modifying the search space of `consec`. Instead of searching for sequences using only the action

trajectories, `consec` can also be used to search for sequences of observations and actions by examining both the observation and action trajectories. As described in Section 6.1.2, these observation and action sequences can be used to create conditionally terminating sequences.

We compared the use of conditionally terminating sequences on several different maze-like gridworlds to the use of action sequences and to the use of primitive actions. Although the agent was able to completely observe the state of its environment, conditioning the sequence's termination based on a specific position in the gridworld would not be as general as if the sequence used a more common set of observations. Instead, the agent's observation was based on the status of the four gridworld squares that it could move to in one primitive step. If there was an obstacle in any of the squares, the sensation for that square was set to one. Otherwise, the agent sensed a zero for that square. These observations were used only by the conditionally terminating sequences and not as part of the agent's state representation as it learned an action-value function.

The RL agent used the maze shown in Figure 6.5 to discover the conditionally terminating sequences. The parameters for the sequence creation algorithm and for learning were the same as with the experiments where the method created action sequences except that support was decreased to 0.85. This was necessary due to the variability in the environment. For this experiment, successful trajectories were defined as those that took 500 steps or fewer to reach the goal. `Consec` waited until 15 such episodes had been experienced before looking for sequences. For comparison purposes, we also used this gridworld and the same parameters to create action sequences that did not conditionally terminate. We used both types of sequences for knowledge transfer. The greedy performance for the agents learning the two types of sequences and for the agent learning with primitive actions only is shown in Figure 6.5B. Neither of the agents discovering sequences was able to do so in time to accelerate learning on this task. However, the sequences were saved for task transfer.

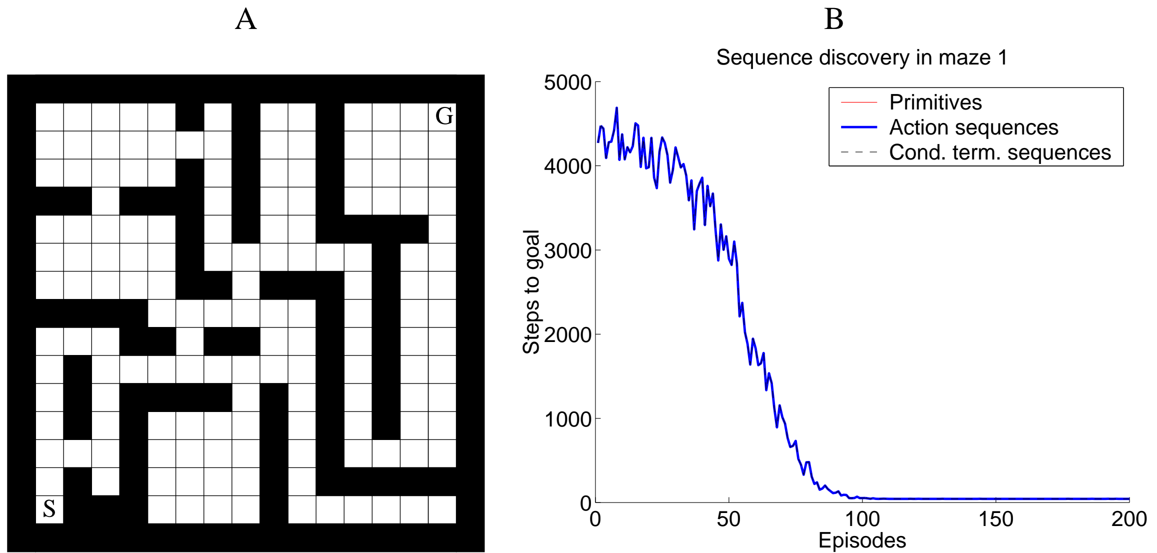


Figure 6.5. A: The maze where the algorithms automatically discovered action sequences and conditionally terminating sequences. Obstacles are shown as filled in squares. B: Comparison of learning while automatically discovering the sequences to learning with primitive actions only.

The reason that the sequences took longer to discover as compared to the earlier task was because this task was much more difficult than the gridworld with no obstacles. Since the sequences were not created in time to accelerate learning on the current task, the main utility of the sequences appears when transferring knowledge from one task to another. To demonstrate this, we reused the sequences learned in the maze shown in Figure 6.5A on the maze shown in Figure 6.6A. The new maze is the same size and shares some of the same structure near the goal region. The agent had again created one option per run for 30 runs of learning in the first maze. As with the task transfer experiments described above, the agent reused both types of sequences in this new gridworld by using the sequences from the corresponding run of the first maze task. The learning parameters remained the same.

We compare the greedy performance of the agent reusing the conditionally terminating sequences to the greedy performance of the agent reusing action sequences and to the performance of an agent using primitive actions only. The results of this comparison are shown in Figure 6.6B. The results of the agent using the action sequences are shown with a thick

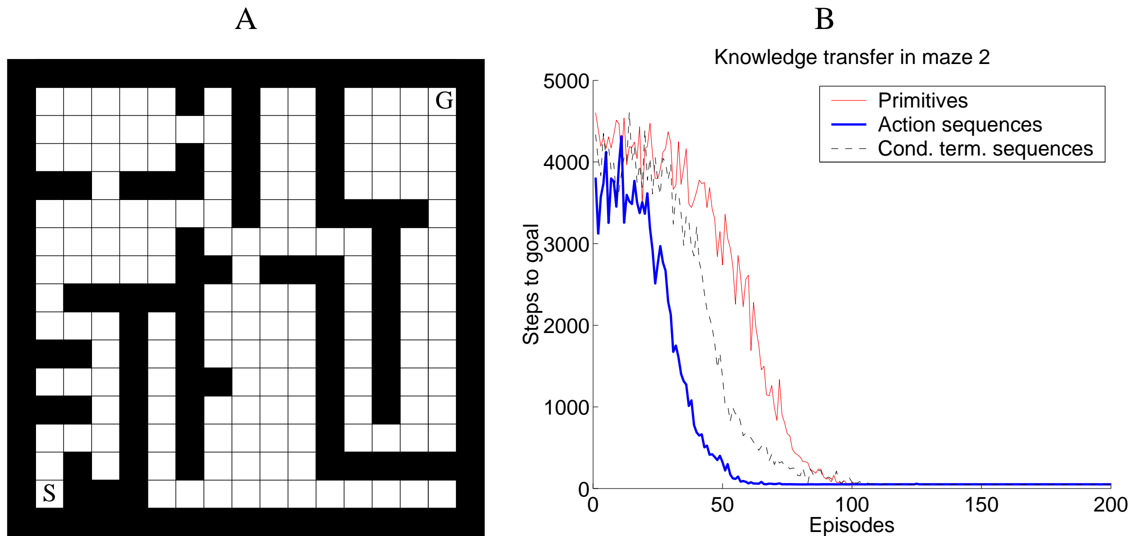


Figure 6.6. A: The second maze world B: Comparison of learning with the automatically discovered action sequences and conditionally terminating sequences to learning with primitive actions only.

line and the results of the agent using the conditionally terminating sequences are shown with a dashed line. The results indicate that both types of sequences were able to significantly accelerate learning over learning with primitive actions only. The action sequences appear to give the agent an advantage over learning with the conditionally terminating sequences, which is surprising. We theorize that this advantage is due to the differences in the input sets of the two types of options, which makes a difference in where each type of sequence can be applied. Since the environment remains the same as in the previous task near the goal region, the action sequences may be quickly drawing the agent to the goal once it enters the input set of the sequences. The conditionally terminating sequences are more general and may apply farther from the goal region, which means the agent still needs to search for the goal. We can examine this hypothesis by using the sequences in a completely different gridworld.

Trying to apply the action sequences in a very different environment highlights another disadvantage of the action sequences, in addition to their property of not responding to or changing with sensory inputs. This shortcoming is the difficulty in correctly applying

action sequences to new tasks with completely different state spaces or representations. The input sets of action sequences are constructed using a specific set of states where the sequence can be initiated. If the state representation changes for a new task, then the agent will have no way of knowing what the correct input set of the sequence should be. The input set of the conditionally terminating sequences depends only on the agent's observations and not on the exact state set. This means that as long as the sensations remain consistent across similar tasks, the conditionally terminating sequences can be used for knowledge transfer.

To examine these issues in more detail, we compared the performance of agents reusing the conditionally terminating sequences and the action sequences from the first maze in a completely new maze. This maze is larger than the first maze (25x25 instead of 15x15) and does not share any structure with either of the first two mazes. The new maze is shown in Figure 6.7A. The agent can easily apply the conditionally terminating sequences because the sensations remain the same for this environment. Although the state space has changed, the representation, e.g. the grid squares, has not. This allows the agent to reuse the action sequences by directly transferring the original input set of the action sequences to the new task.

Figure 6.7B shows the results of learning under these three conditions. Again, the results of the agent using conditionally terminating sequences are shown as a dashed line and the results of the agent using the action sequences are shown as a thick line. In this new environment, the agent with the conditionally terminating sequences outperforms both of the other agents. The agent reusing the action sequences did not perform in a significantly different way than the agent using only primitive actions. The reason for the performance difference between the agents using each of the two types of sequences is as we hypothesized above: the conditionally terminating sequences were more generally applicable and terminate if they observe an unexpected sensation.

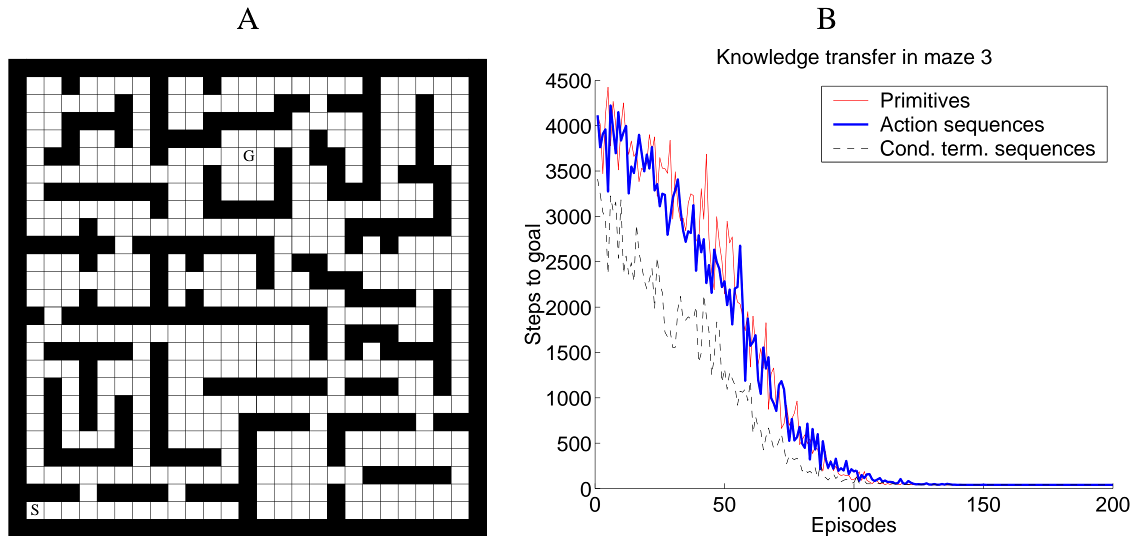


Figure 6.7. A: The third maze world B: Comparison of learning with the automatically discovered action sequences and conditionally terminating sequences to learning with primitive actions only.

6.3 Conclusions

In this chapter, we introduced an algorithm to automatically discover two types of useful action sequences from an agent's experience. For these experiments, the experience came from the online behavior of a learning agent but the next chapter discusses creating sequences from other sources of data. We demonstrated that both types of sequences can be useful for accelerating learning and for knowledge transfer to similar tasks. The conditionally terminating sequences provide a more general method for knowledge transfer to tasks with larger state spaces or with different state representations. The next chapter shows the utility of creating conditionally terminating sequences on a real-world task.

CHAPTER 7

APPLICATION OF SEQUENCE CREATION: INSTRUCTION SCHEDULING

In this chapter, we present the real-world problem of instruction scheduling. We use this domain for illustrating the sequence discovery algorithm introduced in Chapter 6.

Although high-level code is generally written as if it were going to be executed sequentially, most modern computers exhibit parallelism in instruction execution using techniques such as the simultaneous issue of multiple instructions. To take the best advantage of multiple pipelines, when a compiler turns the high-level code into machine instructions, it employs an instruction scheduler to reorder the machine code. The scheduler needs to reorder the instructions in such a way as to preserve the original in-order semantics of the high-level code while having the reordered code execute as quickly as possible. An efficient schedule can produce a speedup in execution of a factor of two or more.

Building an instruction scheduler can be an arduous process. Schedulers are specific to the architecture of each machine, and the general problem of scheduling instructions optimally is NP-Complete (Proebsting, 1998). Because of these characteristics, schedulers are currently built using hand-crafted heuristics. However, this method is both labor and time intensive. With knowledge of how to build algorithms that select and combine heuristics automatically using machine learning techniques, computer architects and compiler designers can save time and money. As computer architects develop new machine designs, new schedulers can be built automatically to test design changes rather than requiring hand-built heuristics for each change. This would allow architects to explore the design space more thoroughly and to use more accurate metrics in evaluating designs.

Machine learning techniques for instruction scheduling could also be used by the end user. Instead of scheduling code using a static scheduler trained on benchmarks when the compiler was written, a user could employ a learning scheduler to discover important characteristics of their code. The learning scheduler would exploit the user's coding characteristics to build schedules better tuned for that particular program.

The sequence discovery algorithm can be useful in both of these situations. The addition of useful sequences can improve the performance of both machine learning algorithms and heuristic searches. This improvement is useful to architects when exploring the design space of a machine. Likewise, if a user tends to have a certain repeatable characteristic to her code that can be extracted into a sequence, the scheduler may be able to produce the same quality schedules more quickly or to produce better schedules in the same amount of time.

7.1 Instruction Scheduling in the Java Virtual Machine

We could conceivably choose to focus on any language that is compiled into machine code on a pipelined machine. However, there is significant overhead in transforming the high-level code of a language into assembly instructions and machine code for a specific architecture. We narrowed our choices by considering only platforms that already existed and that allowed for a new instruction scheduler to be used. This provided greater reliability and stability as well as a potentially broader base of comparisons for our results. In previous work, we demonstrated that a scheduler that used machine learning techniques to make its scheduling choices performed quite well against a commercial C and Fortran compiler and scheduler (McGovern et al., 2002). However, these results were obtained for a processor that has been superseded by newer machines: the Digital Alpha 21064. For the experiments presented in this chapter, we chose to use the open-source Java[1] Virtual Machine from

[1] Java is a registered trademark of Sun Microsystems, Inc.

IBM called the Jikes Research Virtual Machine (JRVM)(Alpern et al., 2000). This choice satisfies both the requirement that we move to a more modern language and architecture and that we use a widely available system. The JRVM is freely available from IBM and is being actively used by a number of researchers. We continue to focus our experiments on RISC-based instruction sets. In particular, we use the Power PC architecture for all of the experiments described in this chapter.

Java is not compiled directly into machine code, like other languages such as C or Fortran. Instead, it is first compiled into a machine independent byte-code format. This byte code is read by a Java Virtual Machine (JVM) and either interpreted directly or transformed into machine code and then executed within the JVM. A high-level view of how Java code is transformed into byte code and then executed by the JRVM is shown in Figure 7.1. We specifically describe the JRVM in this figure. First, a programmer writes Java code into a `.java` file (such as `Hello.java`). This is compiled into machine independent byte code using the `javac` compiler. The JRVM then reads the byte code and performs a number of optimizations and transformations such as instruction scheduling and register allocation. In the JRVM, instruction scheduling takes place after the code has been transformed into Power PC assembly instructions but before its final transformation into machine code and before final register assignments have been made. After the optimizations, the JRVM executes the machine code.

We schedule the assembly instructions within groups of instructions known as basic blocks. A basic block is a set of instructions with a single entry and a single exit point. This means that a basic block cannot contain loops. An example basic block from the "Hello World" program is shown in Figure 7.2A. Our scheduler reorders instructions within each block but cannot add or remove instructions from the block. Although the ability to rewrite the block can also improve the running time, we have previously demonstrated that reordering alone can significantly improve the running time of each block (McGovern et al.,

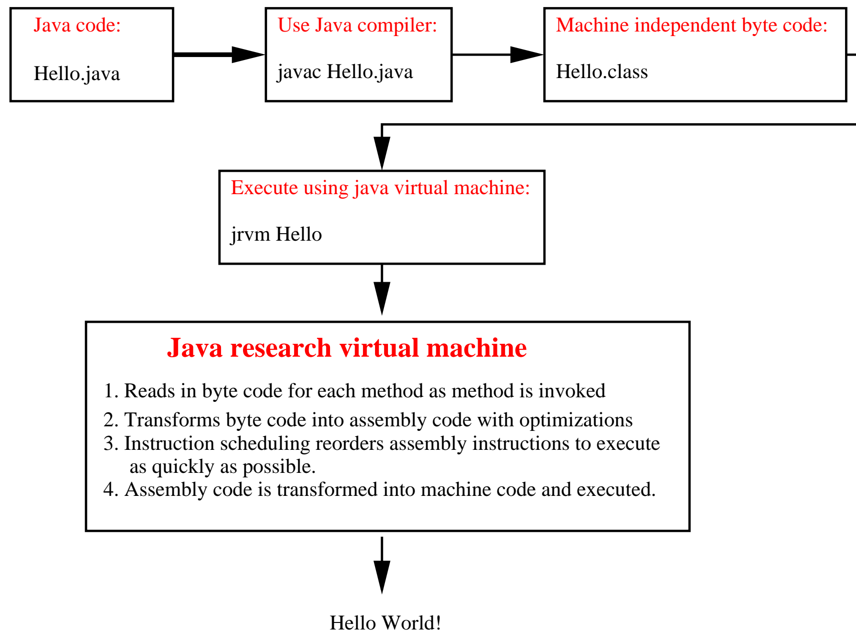


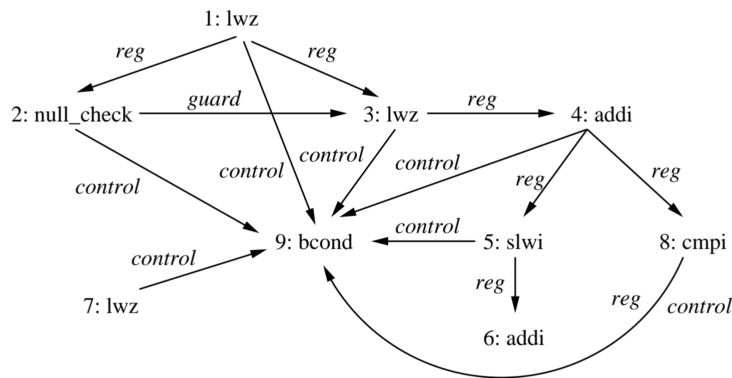
Figure 7.1. Java code is transformed into machine independent byte code and then processed by the Java virtual machine, where our instruction scheduler can reorder the Power PC assembly instructions before they are transformed into machine code and executed.

2002). Limiting the potential number of actions of the scheduler to include only the ability to reorder instructions also enables machine learning methods to work more effectively.

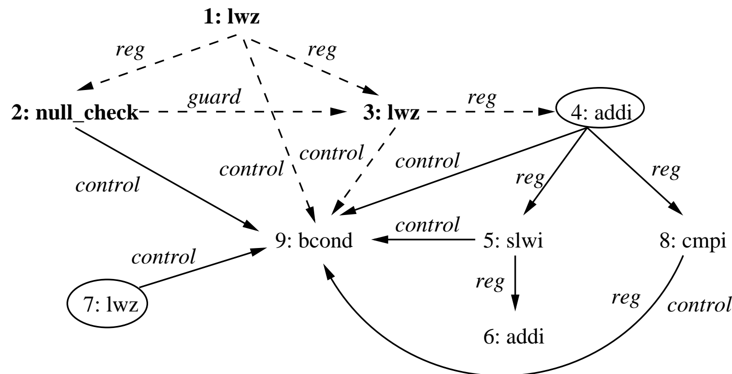
When the scheduler reorders the instructions within a basic block, it needs to do so in such a way as to preserve the in-order semantics of the original Java code. Executing the code in an invalid order could cause a program to return erroneous results or even to fail to run. To ensure that validity of the high-level code is maintained, the JRVM uses a standard approach of representing the constraints with a directed acyclic graph (DAG). The DAG can express constraints based on different types of potential conflicts such as register conflicts or read/write conflicts. An example basic block from the Java program "Hello World" is shown in Figure 7.2A with its corresponding DAG (Panel B). In this example, if the instructions that are shown in bold in Panel C have already been scheduled, then the constraints shown as dashed lines have been satisfied and any of the circled instructions can be scheduled next.

```
1: lwz          R1, 10psi
2: null_check   R2, R1
3: lwz          R3, R1, -4, R2
4: addi         R4, R3, -1
5: slwi         R5, R4, 2
6: addi         R6, R5, 12
7: lwz          R7, JTOC, 268
8: cmpi         R8, R4, 0
9: bctrl        t44c, ppc, ≥
```

A: Sample basic block



B: DAG



C: DAG with some instructions scheduled

Figure 7.2. A: A basic block of Power PC 601 instructions from the Hello World program. B: The DAG for the basic block in panel A. C: Once the instructions in bold have been scheduled, the constraints shown as dashed lines have been satisfied and any of the circled instructions can be scheduled next.

Permutation Schedule a Basic Block(block)

1. Do number of forward passes, p_f, times
2. Initialize the window to the top of the block
3. While (not at end of block)
4. max ← get maximum forward move(window of instructions to schedule)
5. n ← choose action according to policy π
6. move top instruction down n instructions
7. move instructions 2 through n up one slot each
8. move window down 1 instruction

Table 7.1. High-level view of the permutation scheduling algorithm.

7.2 Permutation Scheduling

There are two main approaches for reordering the instructions within the blocks. The first approach is to schedule the block in a greedy manner starting with an empty schedule and continuing until all instructions have been scheduled. This approach, called “list scheduling” in the compiler domain, is the approach that we used in our study of scheduling for the 21064 processor. Although this approach worked quite well, we expected that the second main approach, called “permutation scheduling,” would be better suited to experiments with our sequence discovery method.

In a permutation scheduler, the scheduler is always working with a complete and valid ordering of the instructions. This is different from a list scheduler which produces a completely valid schedule only at the end of each block. Within JRVM, each time the scheduler is called, the basic blocks have already been ordered in a deterministic and valid manner. This ordering is generated from the byte code by translating it into Power PC assembly instructions without optimizations for scheduling. The permutation scheduler reorders the instructions while maintaining the validity of the schedule at each step.

Table 7.1 summarizes how our permutation scheduler reorders a given basic block. There is no limit on the number of instructions in a basic block, which means that an unrestricted permutation scheduler would need to search arbitrarily deep at each step. We

limit the search by allowing the scheduler to consider only permutations within a moving window. The size of the window is specified as a parameter.

The scheduler starts by initializing the window of instructions to be at the start of the basic block. At each step, the scheduler consults the DAG to determine the maximum number of instructions that the top instruction can be moved downward within the window. Because the validity of the schedule is maintained at every step, if the top instruction can be scheduled n instructions later, then any of the scheduling slots from 1 to $n-1$ are also valid choices. Once the scheduler knows how many choices are available for moving the top instruction, it chooses an action using its current action selection policy, π. This policy can be created using heuristics or machine learning techniques. We demonstrate the use of both approaches in this dissertation. The scheduler chooses an action, n, which represents the number of instructions that the top instruction should be moved past. The top instruction in the window is moved down n instructions and the second through the nth instructions in the window are each pushed upward one instruction. The window is then moved down one instruction and the process is repeated until the window reaches the end of the block. This entire method is repeated as specified by a parameter to ensure that the instructions have adequate chances to move within larger blocks. The selection policy, π, is described further in Section 7.6.

7.3 Evaluating the Basic Block Schedules

Each of the different selection methods implemented by the policy π can potentially produce different schedules for the same basic block. In order to determine which methods perform better than others and to provide feedback to machine learning techniques, we need a method for evaluating a given ordering of a basic block. While the actual execution times for each block provide the most accurate estimate of what a user would observe from her scheduled code, these execution times are difficult to obtain. Not only is it difficult to calculate the exact timings for each basic block within a program but the machine must

be put into single user mode and accounting must be made for the overhead of the JRVM and of the scheduling method versus the scheduled code. Obtaining the evaluation of each schedule in this manner would be cumbersome and would make it very difficult to provide timely feedback to machine learning schedulers. Instead, we implemented a simulator inside the scheduler that could estimate the execution times for any given ordering of a set of instructions.

The timing simulator gives the predicted execution time for a set of instructions executing on a particular Power PC architecture. We designed the simulator to be easily extensible and we have implemented it for four Power PC architectures: the 601, 603, 603e, and the 604. Each schedule is evaluated assuming that all resources and functional units are available at the start of the basic block and that all memory references take a constant amount of time. Although these assumptions are not realistic for the actual execution of the program, we used a very similar simulator in our experiments with the 21064 processor and demonstrated that the results obtained from such a simulator are comparable to the actual execution times (McGovern et al., 2002). For learning and evaluation purposes, it is important to note that although the estimated block timing may not correspond exactly with the actual timing of a block, the relative timings of different orderings of the same block of instructions should be maintained. Our system does not simulate the actual execution of the code but instead relies on published timing data for each assembly instruction on each of the different architectures. The DAG is used to enforce the constraints between the instructions.

The timing simulator starts by initializing its set of functional units to be available with all resources free. Each functional unit corresponds to a functional unit for the specified architecture such as an integer or a floating point unit. On each clock cycle, the simulator sends multiple instructions to their preferred functional units. If an instruction can be executed by more than one functional unit, it is sent to the one with the least delay in starting that instruction.

Once the instruction has been passed to a functional unit, the unit accesses the timing data for the instruction. Each instruction has two timings associated with it. The first is the number of cycles that the instruction takes to fully issue, and the second is the number of cycles until the instruction has completed its execution and the results are available to other instructions. Before executing the instruction, the functional unit checks the DAG to ascertain at what clock cycle all of its constraints will be satisfied. If necessary, the functional unit is stalled until the instruction can begin to execute. The functional unit updates all children of the current instruction in the DAG with the earliest clock cycle that they can begin to execute.

Once all instructions have been processed by a functional unit, the timer returns the total number of clock cycles that the instructions took to execute. This timing can be used by the heuristic schedulers or the learning methods.

7.4 Performance Metric

Once we have predicted execution times for all blocks in a program, we can use the times to compare the performance of the different scheduling methods. To do this, we calculate a weighted simulated execution time of each basic block in the program, where the weight for each block is the number of times that the block is executed. This is measured from prior execution of the program. We compare performance with a deterministic heuristic ordering of the code that we denote BASE. This heuristic is described in Section 7.6. More precisely, the performance measure for program P is:

$$\text{ratio}(P) = \frac{\sum_{b \in P} (\text{\# executions of } b)(\text{time to execute } b \text{ under ordering } \pi)}{\sum_{b \in P} (\text{\# executions of } b)(\text{time to execute } b \text{ under ordering BASE})} \tag{7.1}$$

where b is a basic block. This performance ratio indicates the speedup or slowdown of executing the code ordered under policy π compared to executing the code ordered using the BASE heuristic. If the newly reordered code executes more quickly than the BASE

ordered code, then the ratio will be less than one; otherwise, the ratio will be greater than one.

7.5 Benchmark Programs

For the results discussed below, we used the SPEC Java benchmark suite. This suite consists of seven[2] different Java applications each of which focuses on different aspects of a task such as databases or a Java compiler. These programs contain $35,238$ basic blocks of which 90% $(31,791)$ have 12 instructions or less. However, these small basic blocks account for only 48% of the execution time of the programs overall. We schedule basic blocks in sizes of 100 instructions or less. If a block is bigger than this (there are only 70 such blocks), it is broken into blocks of 100 instructions or less. These larger blocks are combined before the basic block timer provides their estimated execution times. This process does not significantly affect the overall running time of the program while significantly increasing the speed of the scheduler.

7.6 Heuristic Schedulers

In addition to the policies learned using RL, we implemented several heuristics for π. The first is the deterministic heuristic that we use as a basis of comparison for our other schedulers. BASE was derived from prior experience with the commercial scheduler on the Digital Alpha 21064 processor. BASE chooses each action based on two features that are calculated for each instruction: `wcp` and `etime`. The first feature, `wcp`, is the weighted critical path, which is the height of the instruction in the DAG with the edges of the DAG weighted by latency of each instruction. The second feature, `etime`, is the earliest time that the instruction can begin to execute if it is scheduled in its current slot. This feature is calculated by finding the minimum number of cycles that the instruction must stall in

[2]We use only the singly-threaded SPEC benchmarks.

the current schedule. At each choice point, BASE chooses the instruction with the smallest `etime`, i.e., the instruction that can start the soonest. If there is a tie, BASE chooses the instruction with the largest `wcp` value. If both features are valued equally for the two instructions in the window, BASE makes no changes and proceeds to the next choice point. Although this heuristic is simple, it has surprisingly good performance, as shown in Section 7.8. This and its simplicity make it a useful scheduler to compare with the performance of the other schedulers.

We also implemented several heuristic schedulers that did not rely on calculating specific features for the instructions being considered at each choice point. These heuristic schedulers require less computational overhead than the ones using features and can provide a useful comparison. The random scheduler, denoted RANDOM, makes all scheduling choices in a uniformly random manner. Although RANDOM is not a good strategy in general, it is a useful baseline. The heuristic scheduler that we denote FIRST always chooses the first available move. This means that FIRST always chooses a move of one whenever a move is available, regardless of the window size. A related heuristic scheduler always chooses the farthest available move. We call this scheduler FARTHEST.

Although these heuristic schedulers do not require significant computational effort to make their choices, a better strategy would not choose moves blindly but rather make informed choices based on features describing the state of the schedule. However, computing features can be computationally expensive. We implemented a hill-climbing scheduler, which we denote HILL. This scheduler simulates the one-step consequences of taking each available move and chooses the move that provides the best improvement in running time. If no improvement is possible within the current window, HILL leaves the schedule unchanged and moves the window appropriately. This scheduler is guaranteed to perform at least as well as the original ordering of the code but it likely to find faster schedules because of the extent of its search. Because HILL makes all choices in a greedy manner, it may not perform optimally. Although it is computationally expensive, hill climbing is useful for

approximating a bound on the best performance of the scheduler. A related heuristic was denoted BASEHILL. In this case, the scheduler first reordered the blocks using the BASE heuristic and then it used HILL to make further adjustments. We can also use the hill-climbing approach to approximate a bound on the worst possible schedules by climbing in the opposite direction, i.e., by taking the worst move at each step. We implemented this heuristic as well, denoted DOWNHILL, purely for comparison purposes and not because we feel it is a useful method for instruction scheduling.

The last heuristic scheduler that we implemented was an optimal scheduler, denoted OPTIMAL. This scheduler searched the space consisting of all valid schedules and returned the best schedule for each block. This search is exponential in the size of the block. Empirical tests showed that it was only feasible on blocks of size 12 or less. It takes an impractical amount of time to search through all valid schedules for larger blocks. Instead, we use BASEHILL to approximate the best schedule for the larger blocks. The OPTIMAL heuristic can also be used to search for the worst possible schedules. We denoted this search SUBOPTIMAL. We used the DOWNHILL heuristic to estimate the worst running times for the blocks of size 13 or more.

7.7 An RL Scheduler

We created a scheduler that learned the best actions to take at each choice point using RL. This section describes the design of the RL scheduler. The number of states needed for a complete description of a basic block and the current status of the processor is large. A basic block has no limit on the number of instructions that it can contain as long as the single-entry and single-exit constraints are satisfied. A description of the exact state of the processor would be quite large because it would include everything from the status of each functional unit, the state of each register and memory cache on the machine, and the current ordering of the basic block. Because of the size, we use a feature-based representation to approximate the state. The features include several instruction-specific features as well

Feature name	**Explanation**
`bblen`	Length of the basic block
`passNum`	Which pass iteration is the scheduler currently on? There are *n* forward passes.
`latency(i)`	Number of cycles that instruction *i* will take to execute.
`wcp(i)`	The height of instruction *i* in the DAG (the length of the longest chain of instructions dependent on this one) with the edges weighted by latency of each instruction.
`children(i)`	The number of children that instruction *i* has in the DAG.
`etime(i)`	The earliest time that instruction *i* can execute if it is scheduled now.
`onwcp(i)`	Is instruction *i* on a weighted critical path from the root to the end of the DAG?
`iclass(i)`	Instruction *i*'s preferred functional unit (e.g., Integer, Floating Point, Branch, etc)
`unit blocked(i)`	Is the preferred functional unit for instruction *i* blocked?

Table 7.2. Features used by the RL scheduler to approximate the state space.

as features that are based on the state of the basic block and processor. The features are summarized in Table 7.2.

Although these features can be used to summarize the state of the basic block and processor, they suffer from some of the same problems described above. For example, the feature `wcp` can take on a potentially infinite number of values depending on the size and configuration of the basic block. However, previous work on instruction scheduling on the DEC Alpha processor demonstrated that certain of the actual feature values did not matter as much as the results of comparing the values for multiple instructions. This means that instead of using the actual values for `wcp`, `latency`, `children`, and `etime`, we calculate the compound features σ(`wcp(i1)`, `wcp(i2)`), σ(`latency(i1)`, `latency(i2)`), σ(`children(i1)`, `children(i2)`), and σ(`etime(i1)`, `etime(i2)`) where:

$$\sigma(a,b) = \begin{cases} 1 & \text{if } a > b \\ 0 & \text{if } a = b \\ -1 & \text{if } a < b \end{cases}$$

Derived Feature name	**Explanation**
`bblen`	Length of the basic block
`passNum`	Which pass iteration is the scheduler currently on? Has *n* possible values where *n* is the number of forward passes.
σ(`latency(i1), latency(i2)`)	Will instruction *i*1 take more, fewer, or the same number of cycles to execute as instruction *i*2?
σ(`wcp(i1), wcp(i2)`	Does instruction *i*1 have a bigger, smaller, or equal value of wcp as instruction *i*2? Instructions with larger values of wcp are often better to schedule first to reduce future instruction dependencies.
σ(`children(i1), children(i2)`)	Does instruction *i*1 have more, fewer, of the same number of children in the DAG as instruction *i*2?
σ(`etime(i1), etime(i2)`)	If instruction *i*1 and *i*2 were each scheduled to be executed next, which would have the smaller delay before it could begin to execute?
`onwcp(i)`	Is instruction *i* on the weighted critical path from the root to the end of the DAG?
`iclass conflict(i1, i2)`	True if instructions *i*1 and *i*2 prefer the same functional unit, and false otherwise.
`unit blocked(i)`	Is the preferred functional unit for instruction *i* blocked?

Table 7.3. The set of derived features used to create the tile coding for the RL scheduler.

The list of features derived from the original feature set is summarized in Table 7.3.

Several of the derived features are calculated based on the corresponding feature values for two instructions at a time. The exact instructions compared depend on the current action being considered. For action *n*, the RL scheduler used the top instruction in the window and the nth instruction. For example, if the window size is three and the window contains the instructions `a`, `b`, and `c`, then for the `no-swap` action, the derived features using two instructions would use instruction `a` for both instructions. Action `swap 1` would use instructions `a` and `b` and action `swap 2` would use instructions `a` and `c`.

We used a CMAC, or tile coding, as the function approximator for the action-value function. The tile coding used the set of derived features to create a coding with 51 layers and 29,472 total tiles. The exact composition of the tile coding is given in Appendix B.

There are configurations of instructions where the scheduler has no action choices available. The RL scheduler learns using only the choice points and does not use configurations of instructions where there is no action choice. If an action is chosen and the resulting schedule has no moves available, the RL backup does not occur unit the next choice point is reached. We use Q-learning to learn the action values. Each transition from one choice point to the next is treated as having occurred in a single time step.

The reward at every step except the last was zero. Once a basic block was completely reordered, the final reward was:

$$r(b) = -10\left(\frac{\text{estimated running time of } b \text{ using the RL generated ordering}}{\text{estimated running time of the original ordering of } b}\right)$$

where b is a basic block. The estimated running times are generated using the basic block timing simulator described above. The overall performance measure for a program weights the ratio of running times by the execution frequency of each block. Since this is not known in advance, we use only the ratio of running times for each block with a constant weight of -10. The negative component of the weight is necessary to give the scheduler a larger reward when a block is reordered to execute more quickly.

7.8 Experimental Results

The instruction scheduling task provides us with a large and complex test bed for the sequence creation mechanism introduced in Chapter 6. We use the scheduling task to explore several different aspects of automatically creating and using sequences. One question that we investigate is what types of behaviors produce the best sequences and why those sequences are better than others. We can create different sets of sequences to examine

this by using the variety of heuristic schedulers that we have implemented to generate the trajectories for the sequence creation algorithm. Using these sequences, we examine the question of what sequences are better than others in two ways. First, we observe the effect of the different sequences on the exploration of the scheduling agent. Second, we test our hypothesis that the better sequences can enable the hill-climbing heuristic to escape from local maxima.

Creating a machine learning scheduler that scheduled as well as the best heuristics proved to be difficult for several reasons. One of the most likely reasons was that the state was only partially observable. Related to this, if two blocks were mapped to the same set of observations but the action choices were not the same, the RL scheduler would average the two sets of choices, which will not work as well as the heuristics that search locally.

The last set of experiments focused on knowledge transfer. In addition to the potential for improving performance within a task, creating sequences is useful because it enables knowledge transfer to similar tasks. With this in mind, we tested whether sequences were able to improve performance across a related set of Power PC architectures.

7.8.1 Performance of Heuristic Schedulers without Sequences

Before examining how sequences affected the performance of the various scheduling algorithms, we briefly discuss the performance of the heuristic schedulers.

We used a window size of two for all of the results presented in this chapter. Although a window of two instruction may seem small, with a sufficient numbers of forward passes in the permutation scheduler, any schedule is reachable from any other schedule for any window size. We chose six forward passes for all of these experiments by empirically testing how many passes were necessary to achieve a steady state using the BASE heuristic. After six forward passes, BASE had converged and made no further changes to the ordering of the blocks.

Benchmark	Ratio of BASE to original ordering
Compress	0.926
Jess	0.973
Db	0.973
Javac	0.975
Mpegaudio	0.978
Raytrace	0.935
Jack	0.976

Table 7.4. Performance of the BASE scheduler as compared to the original ordering for each benchmark.

The first heuristic that we examined is BASE. This heuristic is used to provide a performance baseline for the other heuristics. We compare the performance of BASE to the performance of the original ordering for each of the benchmark programs. The results of these comparisons are shown in Table 7.4. These results indicate that the running times of the code ordered using BASE will be faster than the running time of the original ordering of the code for each of the benchmarks. This performance combined with the simplicity of the BASE heuristic, together with the fact that it is easily implemented, makes BASE a better basis of comparison for the other scheduling methods than the original ordering.

Before analyzing the results of other experiments, we examine the space of possible performance ratios for each benchmark. We used a combination of the heuristics and optimal search to estimate both the best and the worst possible running times for each program. To estimate the best possible times, we used the optimal scheduler on all blocks of size 12 or less. For larger blocks, we estimated the best running times using our best performing heuristic: BASEHILL. To estimate the worst possible running times, we used the suboptimal search for the blocks with 12 instructions or less. The running time of the larger blocks was estimated using the DOWNHILL heuristic. The results of this comparison are shown in Figure 7.3. The exact numbers are given in Appendix B.

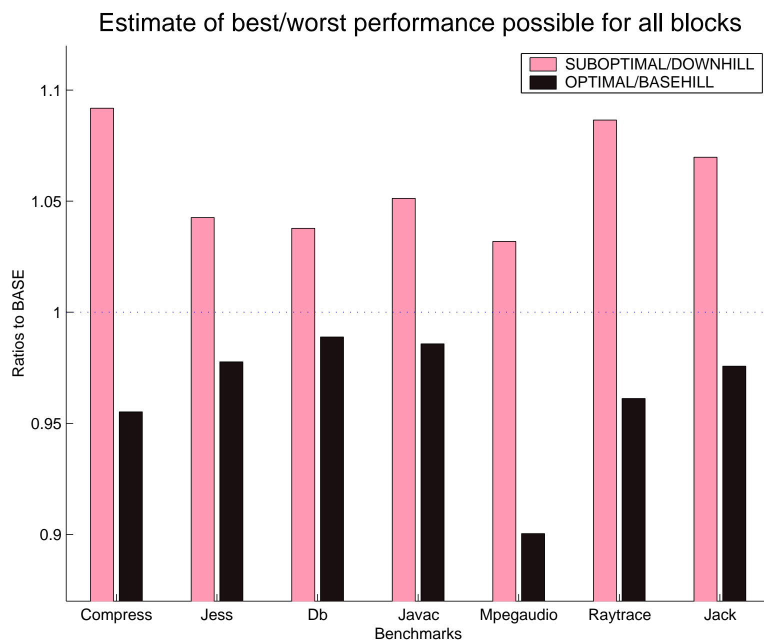


Figure 7.3. Comparison of the estimated best and worst running times for each benchmark. These numbers are for the Power PC 601 processor. The dashed line at 1.0 indicates a running time exactly equal to BASE. The bars above this line represent benchmarks that perform more slowly than under the BASE ordering while the bars below the line represent an ordering of a benchmark that would run faster than BASE.

These results are interesting for several reasons. First, it seems that the most that a scheduler can do is slow a benchmark down or speed it up by approximately 10%. This seems to contradict our original statement that a good scheduler can make a difference in the running time of a program by a factor of two or more. However, we made that statement from our experience with the scheduler on the Digital Alpha 21064 processor. It is possible that we need other benchmarks or that the Power PC 601 architecture is different enough from the 21064 that this speedup is not as easily achieved. It is also possible that if we could truly optimally and non-optimally schedule all blocks, these numbers would change dramatically. Another conclusion that can be drawn from these results is that some of the benchmarks are inherently easier/harder to schedule than others. For example, `compress` has approximately a 15% span between the estimated best and worst running times whereas `db` has only a 6% span.

We also examine the performance of the other heuristics. Figure 7.4 shows the comparison of RANDOM to FARTHEST. The estimated optimal performance is also repeated on the graph for comparison. The results for RANDOM are averaged over 30 runs, each starting with a different random seed. As the results show, FARTHEST performs worse than RANDOM for each of the benchmarks except `javac`. However, the differences are not that large. Neither heuristic is able to outperform the BASE heuristic except on the benchmark `mpegaudio` where RANDOM is 1.6% faster than BASE. From the results shown in Figure 7.3, we know that `mpegaudio` is an easier benchmark to schedule. The exact performance ratios are given in Appendix B.

The next set of heuristics that we compare is HILL and BASEHILL. This comparison is shown in Figure 7.5. The estimated optimal performance is repeated on the graph as well. Both heuristics performed quite well compared to BASE. The BASEHILL heuristic always performed at least as well as HILL and, for the benchmarks `mpegaudio` and `compress`, reordering the schedule using BASE before hill climbing makes a big difference. This is not the case for the other benchmarks. The fact that neither hill climber was able to achieve

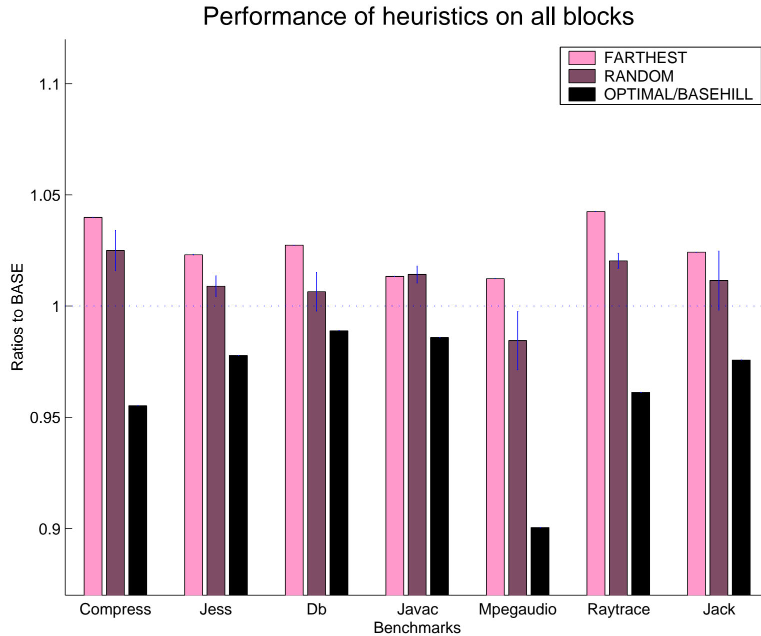


Figure 7.4. Comparison of the FARTHEST heuristic to the RANDOM scheduler for each of the seven benchmarks. RANDOM is averaged over 30 different runs. The estimated optimal performance is also included. The thin lines on the RANDOM bars show the standard deviation above and below the mean.

optimal performance demonstrates that there are local maxima in this problem that can trap hill-climbing schedulers.

Summary and Discussion

We examined the performance of each of the heuristic schedulers using only the primitive actions of `swap` and `no-swap`. We used an optimal search approach combined with the BASEHILL and DOWNHILL heuristic schedulers to approximate the best and worst performances that a scheduler could achieve for the Power PC 601 processor.

Although the HILL heuristic performs quite well, it is clear that this task has local maxima that can hinder the performance of a one-step greedy approach. We examine how the automatically created sequences can improve the performance of HILL in a later section.

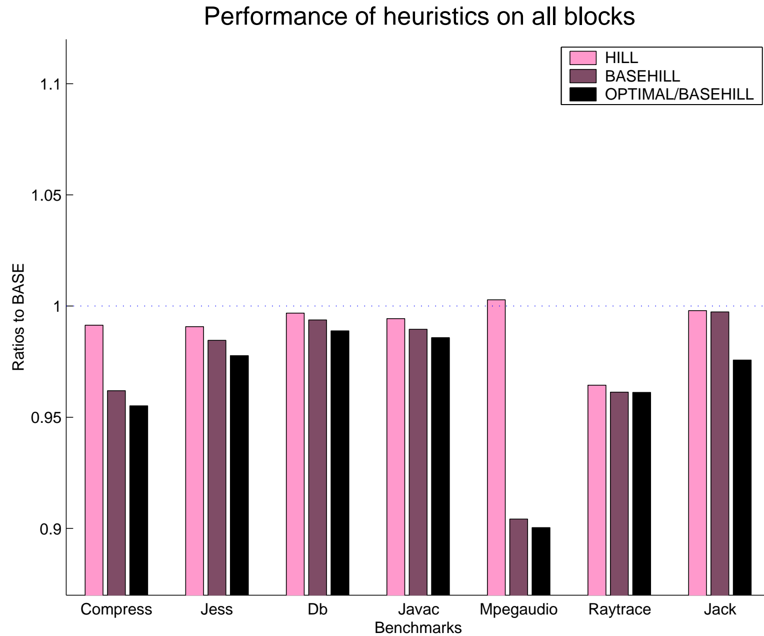


Figure 7.5. Comparison of the HILL heuristic to the BASEHILL scheduler and the estimated optimal scheduler for each benchmark.

7.8.2 Discovering Sequences

The next experiments focus on creating useful sequences from the different behavior trajectories. The sequences were created using the algorithm introduced in Chapter 6. The results of using the sequences in the scheduler are presented in later sections.

One of the questions that we wanted to examine with the experiments in this chapter was which types of behavior trajectories were best for creating useful sequences. With this in mind, we generated eight different sets of trajectories from which to grow the sequences. Five of the trajectory sets came from the behavior of a heuristic scheduler and three from the behavior of an RL agent. The five heuristic trajectory sets used the five heuristic schedulers: HILL, RANDOM, DOWNHILL, BASE, and FARTHEST. Each heuristic scheduler scheduled each of the seven benchmarks and we collected observations and actions at each choice point. The RL agent generated three sets of trajectories by training on different types of data. In the first case, the RL agent trained exclusively on the benchmark `compress`. In the second case, an RL agent trained exclusively on the benchmark `raytrace`. The

last set of trajectories generated by an RL agent came from an agent training on all of the benchmarks. We denote this set of data CROSS. For each of the learning schedulers, after each training run, the scheduler greedily evaluated the current value function and saved its greedy behavior for the sequence creation algorithm.

Each trajectory was composed of the series of observation and action pairs from each choice point visited while scheduling the program. Because we used a window size of two, the two action choices were `swap` and `no-swap`. The observations at each choice point consisted of several local features about the instructions within the window and one block-level feature. The instruction specific features collected were: `wcp`, `latency`, `children`, `onwcp`, `iclass`, and `etime`. Each of these is explained in detail in Table 7.2. The last feature observed was the current forward pass iteration index. Each of the instruction specific features was observed for the two instructions in the window.

With this set of features, there are a potentially large number of observations because there are no bounds on the values of the instruction specific features. However, as discussed in Section 7.7, the actual values of `wcp`, `etime`, `latency`, and `children` are not as useful as the results of a comparison of the values between the two instructions in the window. We used this knowledge to transform the set of observations of actual numbers for each feature into comparison values by using the features σ(`wcp(i1)`, `wcp(i2)`), σ(`latency(i1)`, `latency(i2)`), σ(`etime(i1)`, `etime(i2)`), and σ(`children(i1)`, `children(i2)`) from Table 7.3.

With the transformation of the observations there are 194,400 potential observations (six forward passes, four instruction comparison features with three values each, a binary feature, `onwcp`, for each instruction, and ten possible instruction classes for each instruction). When this is combined with the two possible actions, there are 388,800 possible observation and action tuples. Because of the size of the search space, frequent sequences of observations and actions are unlikely to occur, and those that do so will be very specific to certain situations. To counteract this, we used a subset of the available features to

create sequences which should be more generally applicable. The subset used for these experiments was $\sigma(wcp1, wcp2)$, $\sigma(etime1, etime2)$, and `iclass` for both instructions. We chose this subset through empirical examination of the data, which showed that using the remaining features would make the sequences so specific that they would not apply often enough to affect the performance of the scheduler.

We used the algorithm introduced in Chapter 6 to detect and create each set of sequences. For seven of the data sets, a successful trajectory was defined to be the sequence of observation and action pairs from a reordering of a block where the new ordering would execute more quickly than the original one. For DOWNHILL, we defined success to be those trajectories where the reordering made the schedule worse. Each trajectory contained observations from only one block. With this definition of success, the trajectories for each program ranged from trajectories of length 2 to length 2,264. Because of this disparity, the support level for the sequence detection algorithm, `consec`, needed to be set to a low number in order to find any sequences. This could allow spurious sequences to be detected. We further refined the data by defining successful trajectories to contain at least 15 choice points. This made the set of trajectories for each benchmark more consistent in size by focusing on the larger (and harder) blocks and it allowed us to raise the support level.

For these experiments, the static filter accepted only sequences that contained at least one swap action, those that were not too similar to already existing sequences, and those that had less than a maximum number of actions. The first of these requirements, that of containing at least one swap, was necessary because the set of trajectories often contained long sequences of the `no-swap` action from the later passes of the heuristic schedulers (after they had stabilized). However, a sequence of `no-swap` actions is not a particularly interesting sequence and, as such, we filtered it from the action set. To determine the similarity of two sequences, we compared the sequences of actions (`swap` or `no-swap`) and the sequence of observations. If all of the actions and observations were the same for both sequences, the sequences were said to be the same and the static filter rejected the

Sequences	**Support**	θ	λ	**Number of Trajectory sets**
SEQ-HILL	0.4	3.0	0.9	7
SEQ-RANDOM	0.3	3.0	0.9	7
SEQ-FARTHEST	0.6	3.0	0.9	7
SEQ-DOWNHILL	0.1	2.0	0.9	7
SEQ-SIMPLE	0.45	3.0	0.9	7
SEQ-COMPRESS	0.7	7.0	0.9	20
SEQ-RAYTRACE	0.45	7.0	0.9	20
SEQ-CROSS	0.4	7.0	0.9	21

Table 7.5. Parameters for sequence detection and creation for each of the different data sets.

proposed sequence. If they were different in any regard, the static filter could accept the sequence. The last check that the static filter performed was to ensure that sequences had less than a number of actions that was specified as a parameter. This was useful because we wanted sequences to be general and not specific to large blocks, especially because such a large percentage of blocks in the benchmarks were small.

The eight different behavior trajectory sets were used to create eight sets of sequences using the parameters summarized in Table 7.5. We denote each set of sequences by SEQ and the type of data used to generate the sequences. We kept the parameters as consistent as possible across the different data sets. However, both the stochastic and less purposeful heuristic schedulers, RANDOM and DOWNHILL, did not contain any sequences supported at the levels that we used for the other searches. One conclusion we could reach from this is that non-purposeful searches do not provide useful trajectories for creating new sequences. While this may indeed be the case, we were also interested in the utility of the sequences that could be generated from these behaviors. Consequently, we lowered the support levels for these data sets.

We chose a minimum sequence length of two for all of the data sets because even a sequence of two actions can affect the search space of a scheduling agent. Likewise, all newly created sequences contained at most seven actions and we limited each set to contain

at most ten new sequences. The threshold parameters differ across data sets according to how many trajectories were available to the sequence creation algorithm. Sequences are detected only at the completion of each program, which means that the different data sets have different amounts of data available for sequence creation. The exact numbers are shown in Table 7.5. The five heuristic schedulers were run once for each of the seven benchmarks, which generated seven sets of trajectories each. The two specific learning agents, those trained on `compress` and `raytrace`, trained and generated trajectories 20 times on each of their respective benchmarks. The cross trained RL agent trained 3 times on each benchmark to generate 21 sets of trajectories. We chose the threshold values, θ, such that the sequences needed to appear early and persist throughout the data set. Last, we ran `consec` on each set of trajectories for each program independently of the trajectories for the other programs. This means that we reinitialized the trajectory database after each run of `consec`.

The exact sequences created from each set of behavior trajectories are given in Appendix B. The trajectories collected from the HILL and BASE heuristics generated the fewest sequences while the trajectories generated by the more exploratory behavior supported the creation of the maximum number of sequences.

Summary

We created eight different sets of conditionally terminating sequences from the behavior of five heuristic schedulers (RANDOM, HILL, BASE, FARTHEST, and DOWNHILL) and an RL agent learning to schedule using three sets of training data (COMPRESS, RAYTRACE, and CROSS). The next sections examine the effect of each of the sequences on the exploration of a random scheduler and on the ability of a hill-climbing scheduler to escape from local maxima.

7.8.3 Effect of Sequences on Exploration

One reason that appropriate options can dramatically affect the behavior of an agent is the effect that directed multi-step policies can have on the exploration of the agent. In the case of subgoal options, an agent exploring randomly is better able to explore across weakly connected regions of the state space. With sequences, the agent's exploration becomes less uniform and is instead focused on particular regions of the environment where the sequences tend to take the agent (McGovern, 1998b). We hypothesized that the sequences created for the scheduler from the more purposeful behaviors (SEQ-BASE, SEQ-HILL, SEQ-COMPRESS, SEQ-RAYTRACE, and SEQ-CROSS) would affect the exploration of the RANDOM scheduler toward better schedules. Likewise, we hypothesized that the SEQ-DOWNHILL sequences would affect the exploration of the scheduler toward worse schedules. The RANDOM and FARTHEST heuristic schedulers were not purposefully reordering the code toward either better or worse schedules. Because of this, we expected that the performance of the random scheduler with the SEQ-RANDOM and SEQ-FARTHEST sequences will not differ significantly from the performance of the random scheduler without sequences.

In order to test these hypotheses, we allowed the random scheduler to schedule using each set of automatically created sequences as well as the two primitive actions of `swap` and `no-swap`. The effect of the sequences on exploration is seen more clearly when using the random heuristic because other heuristics already direct the search of the scheduler whereas the random scheduler uniformly explores using the available action set.

For these experiments, we compare the average performance of RANDOM using primitive actions only to the performance of RANDOM using each of the eight different sets of sequences as well as the primitive actions. The scheduler could use only one set of sequences at any given time. Since the random scheduler is stochastic, we averaged the results over 30 different runs.

Sequences	Hypothesis of t-test	Benchmarks					
		Compress	Jess	Db	Javac	Mpegaudio	Raytrace
SEQ-DOWNHILL	$>$		√	√	√		√
SEQ-RANDOM	$\neq$	√				√	√
SEQ-FARTHEST	$\neq$	√		√		√	
SEQ-BASE	$<$		√	√	√	√	√
SEQ-HILL	$<$		√		√	√	√
SEQ-COMPRESS	$<$	√				√	
SEQ-RAYTRACE	$<$	√			√	√	√
SEQ-CROSS	$<$	√				√	

Table 7.6. Results of the t-tests comparing the performance of the RANDOM scheduler with sequences to the performance of the RANDOM scheduler with no sequences. The hypothesis for each t-test is shown in the second column. Any hypothesis accepted with a p value of 0.05 or less is checked. The results for the benchmark `jack` are not shown because none of the t-tests accepted the hypothesis on this benchmark.

The results for these experiments are shown in two formats: using bar graphs and using t-tests. We use both types of results in discussing each experiment. The results of all of the t-tests are summarized in Table 7.6. The table lists the sequence set in the first column, the hypothesis used for the t-test in the second column, and the results of the t-test for six of the seven benchmarks in the remaining columns. The results for the benchmark `jack` are not included in the table because none of the t-tests accepted a hypothesis for this benchmark. T-tests can be used to test three hypotheses: that two sample sets are drawn from distributions with different means, that the true mean of one distribution is less than the mean of the other, or that the true mean of one distribution is greater than the mean of the other. We use all three types of tests in these experiments.

In order to test one hypothesis for these experiments, we need to perform seven t-tests in a row, one per benchmark. A potential criticism of using multiple t-tests to test one hypothesis is that if m tests are used with a p value of x, then there is a $1-(1-x)^m$ probability of incorrectly rejecting the null hypothesis. To alleviate this concern, we used the formula $\alpha = 1 - \sqrt[m]{1-x}$ to determine what p value (e.g. α) should be used to correctly reject and accept m t-tests. For these experiments, m was seven because there was one test

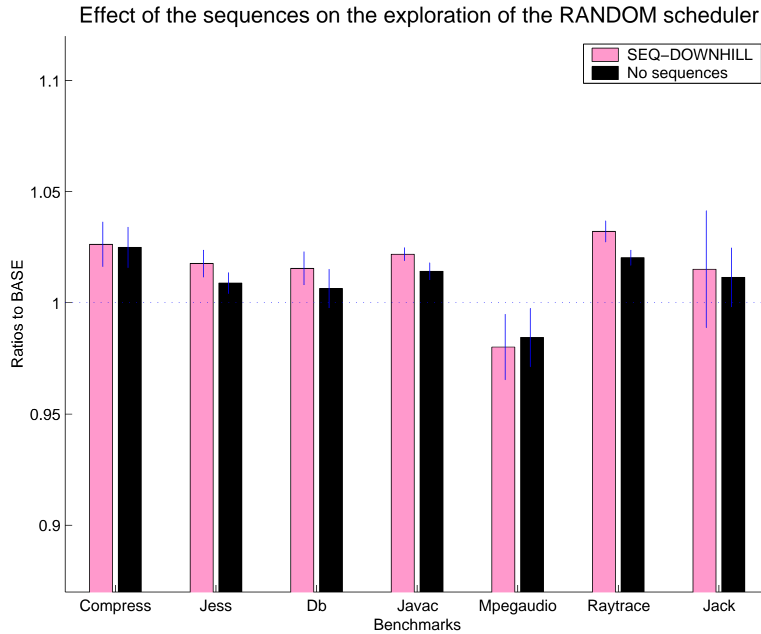


Figure 7.6. Comparison of the performance of the RANDOM scheduler with no sequences to the performance of the RANDOM scheduler with sequences created from the DOWNHILL heuristic. The error bars are one standard deviation above and below each mean.

per benchmark. We wanted the tests to have a certainty level of $p = 0.05$ which meant that we used $\alpha = 0.00028$ for each test. Any test where the hypothesis was accepted is shown with a check mark in the table.

The first hypothesis that we examine is that the SEQ-DOWNHILL sequences will cause the random scheduler to perform worse than if it did not use the sequences. The results of comparing the performance of RANDOM with the SEQ-DOWNHILL sequences to the performance of RANDOM for each benchmark are shown in Figure 7.6. The lighter bars show the results using the SEQ-DOWNHILL sequences and the darker bars are the results with no sequences. Error bars are drawn one standard deviation above and below the means of each. It appears from the graph that the SEQ-DOWNHILL sequences are influencing the exploration of the scheduler toward slower schedules. Table 7.6 shows the results of the t-tests on the hypothesis that the mean performance of the scheduler with the SEQ-DOWNHILL sequences is greater than the mean performance of RANDOM. This is true

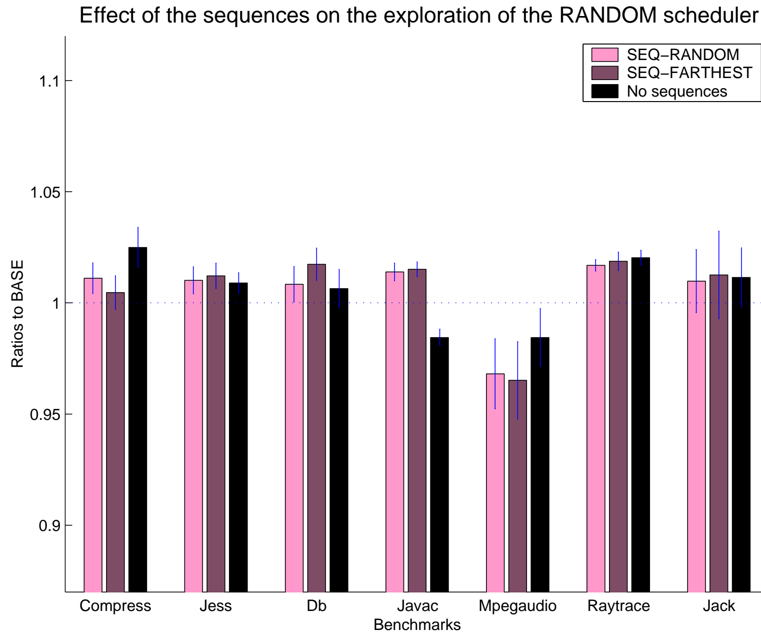


Figure 7.7. Performance comparison between the RANDOM scheduler with no sequences and the RANDOM scheduler with sequences created from the FARTHEST and RANDOM heuristics. The error bars are one standard deviation above and below each mean.

for four of the seven benchmarks, which indicates that our hypothesis was mostly correct. The remaining benchmarks had too large a standard deviation to determine if the means were different.

The next experiment examined the hypothesis that the SEQ-RANDOM and SEQ-FARTHEST sequences would not significantly affect the behavior of the random scheduler. The results of comparing the performance of the random scheduler that used each of these two sets of sequences to the performance of the random scheduler without sequences are shown in Figure 7.7. The results of the t-tests with the hypothesis that the means were not equal are given in Table 7.6. The hypothesis that the means were not significantly different was accepted for four of the seven benchmarks for both sets of sequences. The sequences actually improved the performance on two of the benchmarks (`compress` and `mpegaudio`). For the benchmark `db`, the SEQ-FARTHEST sequences degraded the performance of the random scheduler, and for the benchmark `raytrace`, the SEQ-RANDOM sequences im-

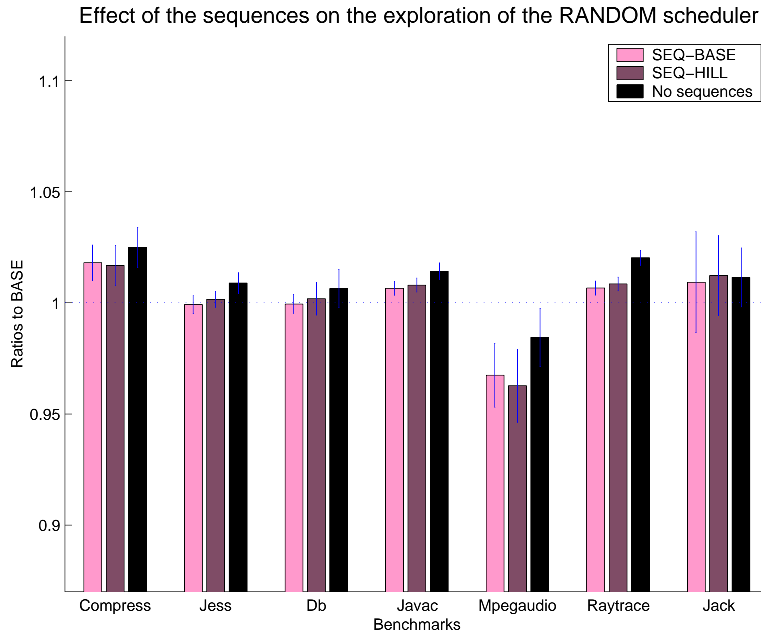


Figure 7.8. The performance of the RANDOM scheduler using sequences created from the BASE and HILL heuristics as compared to the performance of RANDOM with no sequences. The error bars are one standard deviation above and below each mean.

proved performance. Although it is not the case that the sequences did not significantly affect performance of all of the benchmarks, it is clear that these sequences did not uniformly move the exploration of the random scheduler toward better or worse schedules.

The last set of experiments using sequences with the random scheduler investigated whether the sequences created from the more purposeful heuristics would cause the random scheduler to tend to visit better schedules more often. We first examine this hypothesis for the non-learning heuristics BASE and HILL. The performance comparison of the random scheduler with the SEQ-BASE and SEQ-HILL sequences to the performance of the random scheduler without sequences is given in Figure 7.8. The results of the t-tests with the hypothesis that the means of the random scheduler with the sequences is less than the mean of the random scheduler with no sequences are given in Table 7.6. It appears from the graph that this hypothesis is verified for all but the benchmark `jack`, but in fact, although the performance on the benchmark `compress` appears better than RANDOM, it is not a

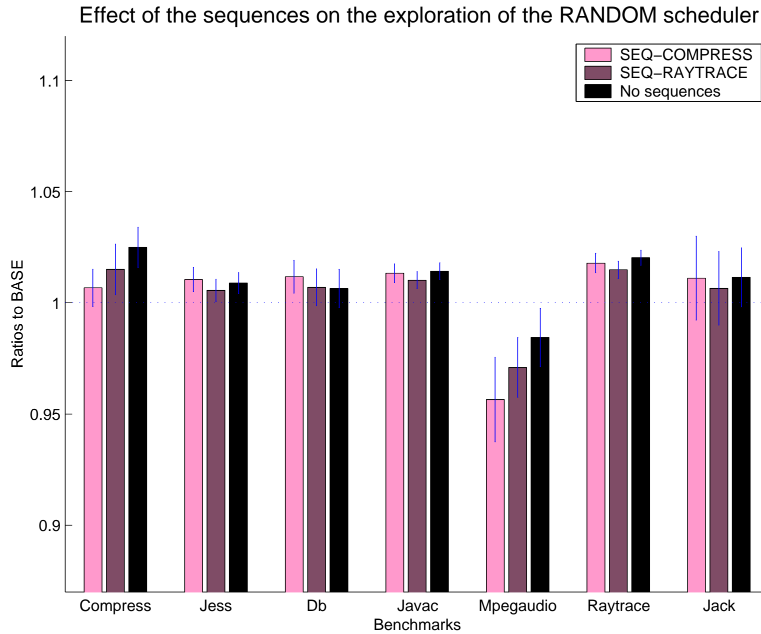


Figure 7.9. Performance of the RANDOM scheduler as compared to the performance of the RANDOM scheduler with the sequences created from the behavior of the RL scheduler that trained on the benchmarks COMPRESS and RAYTRACE. The error bars are one standard deviation above and below each mean.

statistically significant difference. However, our hypothesis that these purposeful behaviors would create sequences that move the exploration toward better schedules is valid for the majority of the benchmarks.

Figure 7.9 shows the comparison of the random scheduler using the SEQ-COMPRESS and SEQ-RAYTRACE sequences to the performance of RANDOM. Table 7.6 again shows the results of the t-tests on the hypothesis that the sequences will improve the performance of the RANDOM scheduler. The SEQ-COMPRESS sequences were able to improve the performance on only two benchmarks but one of these was the benchmark `compress`, which is the one used to generate the sequences. Likewise, the SEQ-RAYTRACE sequences improved performance on the benchmark `raytrace`. Overall, the SEQ-RAYTRACE sequences either improved or maintained the performance of the RANDOM scheduler. The SEQ-COMPRESS sequences did not improve performance as frequently. One

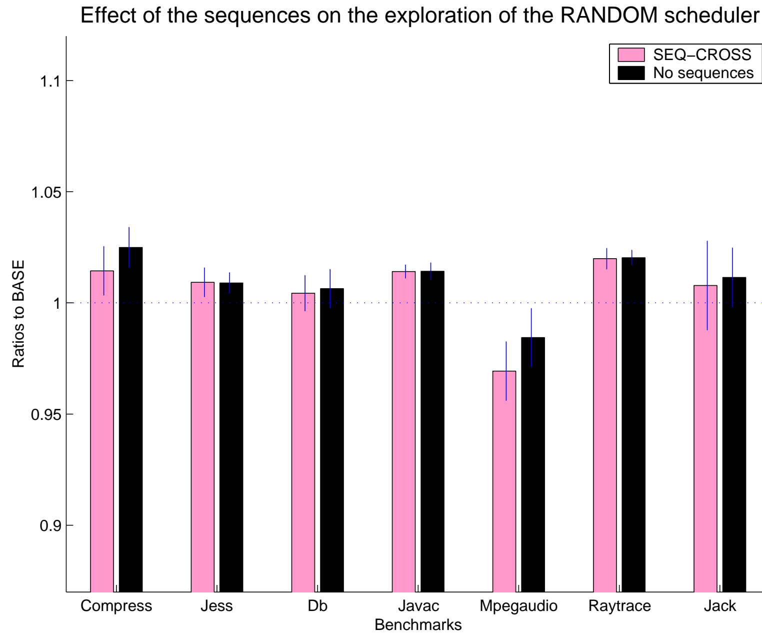


Figure 7.10. Performance of the RANDOM scheduler using the sequences created from the RL scheduler that trained on each benchmark (denoted CROSS) as compared to the performance of the RANDOM scheduler. The error bars are one standard deviation above and below each mean.

possible reason for this is that the benchmark `raytrace` may share more characteristics with the other benchmarks than `compress` does.

The last comparison that we made with the RANDOM scheduler used the SEQ-CROSS sequences. These results are shown in Figure 7.10 and the corresponding t-tests are given in Table 7.6. These results are comparable to the results with the SEQ-COMPRESS sequences. Although the SEQ-CROSS sequences only improved performance on two benchmarks, they did not harm the performance for the other benchmarks.

Summary and Discussion

The experiments presented in this section mostly verified our hypothesis that the sequences created from the more purposeful behaviors would positively affect the exploration of the scheduler while the sequences created from the less purposeful behaviors would negatively affect the exploration of the scheduler. The sequences created from the DOWN-

HILL heuristic generally moved the exploration of the scheduler toward worse schedules. The SEQ-RANDOM and SEQ-FARTHEST sequences did not significantly affect the exploration of the scheduler and the sequences created from the more purposeful data of BASE and HILL moved the exploration of the scheduler to a space with better schedules. The sequences generated from the learning data were the most surprising because they did not generally improve the performance of RANDOM. However, this may be due to poor learning on the part of the RL agent.

We next explore the effect of the sequences on the hill climbing heuristic to determine if the sequences enable the hill climber to escape from local maxima.

7.8.4 Effect of Sequences on the Hill Climbing Scheduler

If the automatically created sequences are useful, they should also be able to improve the performance of the heuristic schedulers. In particular, because the hill-climbing scheduler acts in a greedy manner, it can become stuck in local maxima instead of finding a globally optimal solution. Multi-step action sequences enable the hill climber to look more than one step ahead before choosing the greedy action. While this does not guarantee that the hill climber will be able to escape from local maxima, it increases the probability of doing so by increasing the depth of the search.

We expected that the performance of a hill-climbing scheduler that was able to use the automatically created sequences would be better than the performance of the hill climber without sequences if the sequences were created from the more purposeful behaviors. However, exactly which behaviors would produce the best sequences for the hill climber should be different than for the RANDOM scheduler. This is because exploration is already built into the RANDOM scheduler and it needs only to be directed to the better schedules. The hill climber does not explore stochastically, which means that it needs sequences that can provide a capability of moving away from the slower schedules. If the data used to create the sequences contained no experience at escaping from local maxima, then it is unlikely

that the corresponding sequences will improve the performance of the hill climber. This is especially true for the SEQ-HILL and SEQ-BASE sequences. Also, since the hill climber chose only actions that improved its performance, we further expected that the sequences created from the DOWNHILL and RANDOM data would neither hinder nor help the hill climber's performance. We expected the remaining sets of sequences (SEQ-FARTHEST, SEQ-COMPRESS, SEQ-RAYTRACE, and SEQ-CROSS) to improve the performance of the hill climber.

The results presented in Section 7.8.1 indicated that the hill climber performed well with respect to the optimal scheduler, although there was room for improvement by moving the hill climber away from local maxima. For these experiments, instead of comparing the differences in the performance ratios of the hill climber with the sequences to the performance ratios of the hill climber without sequences, we calculated the percent improvement toward the performance of the estimated optimal scheduler when using sequences. This is a standard technique for combinatorial optimization problems where a known heuristic's performance is already close to optimal and the task is to narrow that difference. We used the following equations to calculate the improvement:

$$\begin{aligned}
\text{improvement(P)} &= [(\text{ratio(HILL(P))} - \text{ratio(optimal estimate(P))}) - \\
&\quad (\text{ratio(HILL with sequences(P))} - \text{ratio(optimal estimate(P))})] \\
\%\text{closer to optimal(P)} &= \frac{\text{improvement(P)}}{\text{ratio(HILL(P))} - \text{ratio(optimal(P))}} \qquad (7.2)
\end{aligned}$$

where P is the program scheduled using the specified heuristic and the ratios are calculated using Equation 7.1.

We compared the performance of HILL with each of the eight different sets of sequences on each of the seven benchmark programs. The numbers reported are from only one run because the hill climber is deterministic. We again present the results in two ways: by using bar graphs and by examining the average percent improvement toward the es-

Sequences	Average % improvement	Standard deviation	Best improvement	Worst improvement
SEQ-FARTHEST	47.63	27.61	90.30	3.37
SEQ-RANDOM	36.28	19.87	69.86	2.71
SEQ-COMPRESS	22.66	16.28	46.31	0.34
SEQ-RAYTRACE	18.69	17.19	46.30	0.19
SEQ-DOWNHILL	5.54	7.45	17.25	-0.35
SEQ-HILL	3.61	9.09	17.25	-10.11
SEQ-BASE	3.61	9.09	17.25	-10.11
SEQ-CROSS	2.43	6.54	17.25	-0.35

Table 7.7. The average percent improvement toward the estimated optimal scheduler for the hill climbing scheduler using the sequences created from each of the eight data sets. The standard deviation and the minimum and maximum improvements are also given.

timated optimal schedules. The latter results are summarized in Table 7.7. The exact improvement for each benchmark and set of sequences are given in Appendix B.

The results of using the sequences with the hill climber were mostly as we predicted with a few exceptions. As we expected, the SEQ-DOWNHILL, SEQ-HILL, and SEQ-BASE sequences did not significantly improve the performance of the hill climber (each had an average of 6% or less improvement). The hill climber using the sequences from HILL and BASE actually performed 10% worse than the hill climber without the sequences on the benchmark `raytrace`. It is likely that the availability of the multi-step actions led HILL to discover local maxima that would not have been reached with single-step actions in this case. Also as expected, the more purposeful but exploratory behaviors of the RL scheduler that trained on the benchmarks `compress` and `raytrace` were able to improve performance by 23% and 17% respectively. Each was able to improve the performance on the benchmark on which it was trained but neither was able to improve the performance on the difficult benchmark `jack`.

The first surprising result was that the SEQ-CROSS sequences failed to improve the performance of the hill climber except on one benchmark, `mpegaudio`. However, the performance of HILL with sequences was consistently better than HILL without sequences

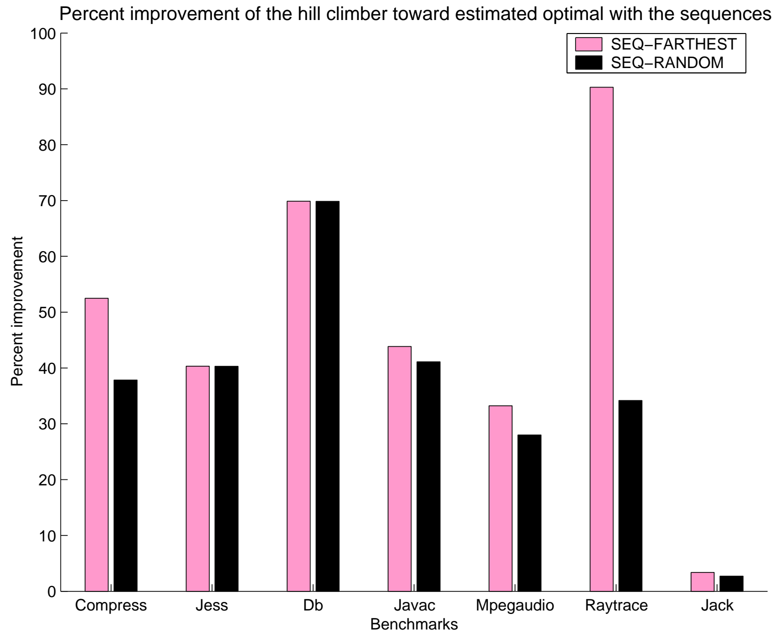


Figure 7.11. Percent improvement toward the estimated optimal schedule for the hill climbing scheduler with sequences created from the FARTHEST and RANDOM heuristics. This improvement is measured against the hill climber with no sequences.

on this benchmark. The second unexpected result was that the best sets of sequences turned out to be SEQ-FARTHEST and SEQ-RANDOM. We show the performance of HILL with these sequences in more detail in Figure 7.11.

Figure 7.11 shows the percent improvement toward the estimated optimal schedules as described by Equation 7.2 for the hill climber with the SEQ-FARTHEST and SEQ-RANDOM sequences. Both sets of sequences are able to improve the performance of the hill climber on all of the benchmarks. The benchmark `jack` seems to be the most difficult to improve but significant progress toward the estimated optimal schedule is evident on the other benchmarks. It is likely that longer sequences are necessary to climb out of the remaining local maxima.

The reason that the SEQ-RANDOM and SEQ-FARTHEST sequences were best for improving the performance of the hill climber is likely two-fold. The first reason is that the SEQ-FARTHEST sequences cause the hill climber to investigate the results of many swaps

in a row. Other sets of sequences do not share this characteristic. Second, although the baseline performance of both of the RANDOM and FARTHEST heuristics is fairly poor, we have pruned the experience used to create the sequences to only those blocks where the heuristics improved the running time. This means that the data used to create the sequences contains more experience at escaping from local maxima than the other data sets. The hill climber can make use of that experience through the sequences.

Summary and Discussion

This section presented results of using each of the eight different sets of sequences with the HILL heuristic scheduler. Two sets of sequences, SEQ-RANDOM and SEQ-FARTHEST, were able to significantly improve the performance of the hill-climbing scheduler toward the estimated optimal scheduler. These particular sequences were created from the behavior of poorly-performing exploratory heuristics. However, by narrowing the experience used to create the sequences to the blocks where the heuristic improved the running time of the scheduler, these sequences can provide HILL with knowledge of how to escape from local maxima.

These results are important for several reasons. Not only have these experiments demonstrated the utility of the automatically created sequences within one task, but they have demonstrated one form of knowledge transfer. The sequences created from the behavior of an RL agent learning on a specific benchmark (SEQ-COMPRESS and SEQ-RAYTRACE) were able to improve the performance of the hill-climbing scheduler on the benchmark on which they were trained as well as on the other benchmarks. The next section examines knowledge transfer across processors.

7.8.5 Knowledge Transfer Experiments

We demonstrated one form of knowledge transfer in the previous experiments by examining how the sequences generated from one benchmark (i.e., SEQ-COMPRESS and SEQ-RAYTRACE) successfully improved the performance of the hill climber on other

Sequences	**Average % improvement**	**Standard deviation**	**Best improvement**	**Worst improvement**
SEQ-FARTHEST	21.70	21.44	61.50	-0.17
SEQ-RANDOM	19.89	21.88	59.62	-0.17
SEQ-COMPRESS	5.03	5.63	13.30	-0.74
SEQ-RAYTRACE	0.89	2.71	7.01	-0.74
SEQ-DOWNHILL	-0.13	0.27	0.00	-0.74
SEQ-HILL	-0.13	0.27	0.00	-0.74
SEQ-BASE	-0.13	0.27	0.00	-0.74
SEQ-CROSS	-0.13	0.27	0.00	-0.74

Table 7.8. The average percent improvement toward the estimated optimal scheduler for the hill climbing scheduler on the Power PC 603 processor using the sequences created from each of the eight data sets on the Power PC 601 processor. The standard deviation, best, and worst improvements are also given.

benchmarks. Although we know that the best scheduler for any processor is specific to the architecture of that processor, we are interested in another form of knowledge transfer for the sequences. In particular, we examined whether sequences could also improve the performance of the hill climber on processors similar to the one for which the sequences were created.

For the experiments described so far in this chapter, we generated the behavior data, created the sequences, and tested the sequences on the Power PC 601 processor. For these experiments, we used the sequences created for the 601 processor on the Power PC 603 processor. The architectures of the two processors are similar although the 603 processor has several additional functional units. We did not expect that the sequences created on the 601 architecture would bring the hill climber on the 603 processor as close to optimal performance as they were able to do for the hill climber on the 601 processor but we expected measurable improvement from the better sequences (i.e., SEQ-RANDOM and SEQ-FARTHEST). Again, we used the performance measurement of the percent improvement toward the estimated optimal schedule described by Equation 7.2.

The results of this experiment are summarized in Table 7.8 with the complete details are given in Appendix B. The results of using the SEQ-RANDOM and SEQ-FARTHEST

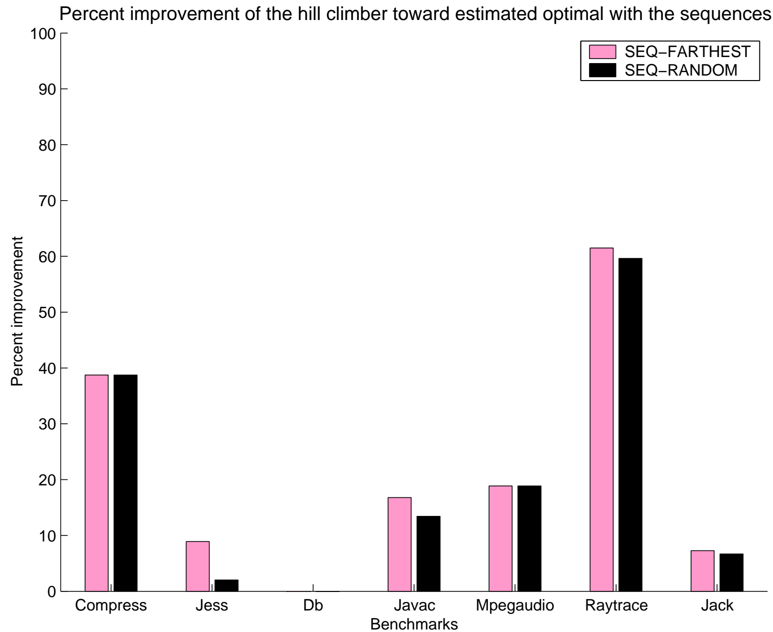


Figure 7.12. Percent improvement toward the performance of the estimated optimal scheduler for the hill climber on the Power PC 603 architecture. The hill climber used the sequences generated for the Power PC 601 architecture from the behavior of the heuristics RANDOM and FARTHEST.

sequences are shown graphically in Figure 7.12. As expected, the SEQ-RANDOM and SEQ-FARTHEST sequences were able to significantly improve the performance of the hill climber toward the estimated optimal performance. The largest improvement was on the benchmarks `raytrace` and `compress`. Neither set of sequences was able to improve performance on the benchmark `db`. Although these improvements are not as large as the improvements observed on the 601 processor, this was expected because of the difference between the respective architectures. These results demonstrate that the SEQ-FARTHEST and SEQ-RANDOM sequences are useful across multiple processors.

A second interesting result of this experiment was that the SEQ-COMPRESS sequences slightly improved the performance of the hill climber overall and specifically on the benchmark `compress`. The SEQ-RAYTRACE sequences proved to be less general for this processor but they were able to achieve a 7% improvement toward optimal on the benchmark

`raytrace`. This demonstrates that the sequences generated from a specific benchmark seem to have captured some characteristic of that benchmark that is true across processors.

Summary and Discussion

These experiments demonstrate that the sequences generated for one processor have not only captured aspects of the benchmarks that are specific to one processor but also have discovered general characteristics that can enable knowledge transfer to similar processors. This is important not only for demonstrating the utility of our approach on a real-world problem but also for any architect who wants to use our algorithm for exploring the design space of a processor. By demonstrating that the sequences generated for one processor are useful on similar processors, we have shown that our algorithm could be used to generate sequences on a general formulation of a processor and then the sequences could be used to improve the performance of each processor in more specific testing.

7.9 Conclusions

In this chapter, we have demonstrated that the sequence growing method introduced in Chapter 6 can create useful sequences in a real-world task. In particular, we created sequences from eight different sets of behavior and used the different sequences to examine the question of what makes some sequences more useful than others in more depth. We demonstrated that sequences created from poorly performing heuristics are not useful to a scheduling agent and can negatively affect the exploration of the scheduler. Likewise, sequences created from the more purposeful behaviors focused the exploration of the random scheduler into better regions of the scheduling space.

We also demonstrated that the behavior of the exploratory heuristics created better sequences for the hill climber by enabling the hill climber to use the experience of escaping from local maxima that the sequences provide. The hill climber significantly improved its performance toward the performance of the estimated optimal scheduler by using these

sequences. We also demonstrate that sequences enabled effective knowledge transfer both within one processor and across similar processors.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

In this dissertation, we have introduced methods for automatically identifying and creating two types of temporal abstractions: subgoals of achievement and action sequences. These temporal abstractions are identified using the accumulated experience of an agent as it interacts with an environment. We have demonstrated that these methods can create temporal abstractions from the behavior of multiple types of purposeful agents. The behavior can be generated by an agent using RL to choose its actions, by an agent using heuristic search, or even from a human tele-operating a robot. By demonstrating that our methods can create temporal abstractions from multiple types of purposeful behavior, we illustrated the generality of our approach.

Subgoal options are created by searching for bottlenecks in the agent's observation space. We formulated this as a multiple-instance learning problem and used the concept of diverse density to detect the subgoals. This approach can create subgoals in continuous spaces as well as in discrete ones. We illustrated our method in several simulated tasks and showed that it can both accelerate learning on the current task and facilitate transfer to related tasks. We demonstrated that similar subgoals can be created from experience generated by an RL agent and from experience generated by humans tele-operating a simulated robot.

Useful action sequences are discovered by searching for actions that are frequently chosen in sequence on successful trajectories. We introduced an efficient algorithm for detecting useful sequences and demonstrated the utility of both action sequences and conditionally terminating sequences in several simulated tasks. These sequences were created

from the behavior of an RL agent. We also demonstrated that useful conditionally terminating sequences can be generated from heuristic search data. We showed that these conditionally terminating sequences can both improve the performance of a real-world instruction scheduler on the processor for which they were created and that they can facilitate task transfer to a similar processor.

Although neither subgoals of achievement nor action sequences will be useful in every task or every environment, we have demonstrated that for many tasks they can both enable task transfer and accelerate learning within a task. These two techniques are not the only possibilities for automatically creating options from an agent's experience in interacting with an environment. Other possibilities include creating rule-based abstractions or finite automata. Although the subgoals discussed in this paper are not identified with the specific criteria of violating the conditions of other subgoals (or allowing the evaluation function to decrease), both Iba's (1989) and Korf's (1985) methods for creating such macro-operators provide ideas for future types of options that might be discoverable by mining the agent's past experience.

Both of our methods also have some potential drawbacks that could be addressed in future work. One potential drawback to this method is the requirement that negative bags cannot contain any positive instances. In the case of more complicated environments, it can be non-trivial to define what constitutes the positive and negative bags. It is desirable to allow the agent to occasionally visit useful subgoals on unsuccessful trajectories while not affecting the results of the diverse density calculations. We are currently investigating how to make the best use of such noisy bags. One such method is to decrease the influence of each negative instance through the width of the Gaussian probability distribution. It may also be possible to give each bag a relative measure of success.

Each of the subgoals presented in this dissertation that used the diverse density method were created by searching in a pre-specified concept space. Maron (1998) presents an extension of diverse density that can identify both a concept and an appropriate scaling

for each dimension of the concept. In practice, we found that scaling the dimensions of the concepts did not work well in our larger test problems because of the noise inherent in the data (in particular, with the simulated robotic tasks). By extending the diverse density approach to allow some noise to occur in the negative bags, the agent may also be able to choose the appropriate dimensions for each concept in a more autonomous manner.

Another possible extension to the diverse density methods could consider more abstract concept spaces in which to compute diverse density. For example, concepts in the form of linearly discriminable regions might allow a robotic agent to detect concepts not expressible as a single point in feature space. We have some preliminary results in this domain that demonstrate that concept spaces other than those introduced by Maron (1998) are both feasible to calculate and can be useful (McGovern and Barto, 2001). Another type of concept that could be useful would be one that preserves some aspect of the temporal associations or dynamics between instances in a bag. For example, it would be better for a robot to learn a concept of "back away from a wall" than "approach a wall." This type of abstraction requires that some of the temporal dynamics of the robot's interaction with its environment are preserved in the concept.

The main drawback to both action sequences and conditionally terminating sequences is that the sequence of actions is fixed once the option has been created. Although conditionally terminating sequences can be stopped if an unexpected state is reached, sequences as introduced in this dissertation cannot adapt over time to a changing environment. It is possible that an alternative representation may allow sequences to be easier to modify. This is an area for future research.

Both methods of option discovery require that the agent must first be able to reach the overall goal using only the given primitive actions (or any options that are pre-defined). This limits the problems to which the methods can be applied. Current research addresses ways to extend the approach so that it can be applied when goal states cannot easily be reached using only primitive actions.

The methods presented in this dissertation identify useful temporal abstractions from an agent's successful interaction with its environment. Using a similar approach, an agent may also be able to identify situations to avoid using its unsuccessful experience in the environment. This type of abstraction would enable an agent to focus its exploration in the more effective and safe areas of the environment. For example, if a robot could quickly learn to avoid walls after bumping into one, it could focus its exploration on learning to navigate within a room.

Iba's (1989) approach to creating macro-operators included a method, called the dynamic filter, for filtering macros that proved to be less useful than expected over time. A similar online method for quickly filtering our automatically created options if they prove to be less useful than expected could help to accelerate learning within a task. For example, preliminary results with a dynamic filter indicate that the few subgoal options discovered in the two-room gridworld described in Chapter 4 that are not in the doorway can be removed by a dynamic filter in favor of subgoals in the doorway as the agent continues to learn. For these results, we used the diverse density method for filtering as well as for discovering the subgoals. The dynamic filter could also be useful in cases in which an option is not as useful on a new task as expected, and in cases where an initially useful option is keeping the agent from discovering an even more efficient path to the goal.

We are also currently investigating the application of our methods to tasks on an actual robotic platform. Although we expect that our methods will prove useful in this domain, physical robots often need extra care to operate robustly and some of the extensions discussed above, such as learning from less experience or using different concept spaces, may be necessary.

Our results suggest that our methods offer a promising approach to one aspect of the challenge of automatic abstraction.

APPENDIX A

DETAILS OF THE TILE-CODING REPRESENTATION USED BY THE SIMULATED ROBOT IN CHAPTER 4

The simulated robot described in Chapter 4 used a CMAC, or tile-coding, over its sensory vector to approximate the action-value function. Table A.1 specifies what sensory readings were used for each layer of the tile-coding and how each dimension of each layer was divided. The number of bins is the number of divisions for the specified variable in the current layer.

Layer	Sensory variable	Bins
Layer 1	Leftmost sonar (#1)	3 (near, middle, far)
	Middle sonar (#4)	3
	Rightmost sonar (#7)	3
	Orientation of object on retina	11
	Is there an object in the gripper?	2
Layer 2	2nd left sonar (#2)	3 (near, middle, far)
	Middle sonar (#4)	3
	2nd right sonar (#6)	3
	Orientation of object on retina	11
	Is there an object in the gripper?	2
Layer 3	3rd left sonar (#3)	3 (near, middle, far)
	Middle sonar (#4)	3
	3rd right sonar (#5)	3
	Orientation of object on retina	11
	Is there an object in the gripper?	2
Layer 4	Middle sonar (#4)	3 (near, middle, far)
	Estimated forward velocity	4
	Estimated rotational velocity	4
	Is there an object in the gripper?	2
Layer 5	Leftmost sonar (#1)	3 (near, middle, far)
	Middle sonar (#4)	3
	Rightmost sonar (#7)	3
	Estimated forward velocity	4
	Estimated rotational velocity	4
	Is there an object in the gripper?	2
Layer 6	Width of object on retina	7 (logarithmic)
	Orientation of object on retina	11
	Is there an object in the gripper?	2

Table A.1. Details of the simulated robot's tile-coding used to approximate the action value function.

APPENDIX B

DETAILED RESULTS FROM THE INSTRUCTION SCHEDULING TASK PRESENTED IN CHAPTER 7

This appendix describes the sequences created for the instruction scheduling task described in Chapter 7 and the description of the tile-coding used by the RL scheduler. We also include the detailed results for each experiment described in Chapter 7. These include the performance ratios for each heuristic and benchmark without the use of sequences as well the results of adding sequences to the random and hill climbing schedulers.

B.1 Tile-coding Representation Used by the RL Scheduler

The tile-coding used by the RL scheduler for all of the learning experiments presented in Chapter 7 is given in Tables B.1 to B.7.

B.2 Conditionally Terminating Sequences Created for the Scheduler

We used eight different sets of behavior trajectories to create the eight sets of sequences. The behaviors were generated from the five heuristics: RANDOM, DOWNHILL, BASE, HILL, and FARTHEST and from three different training sets for RL: COMPRESS, RAYTRACE, and CROSS. These are described in more detail in Chapter 7.

Tables B.8 to B.15 show the conditionally terminating sequences created from each of the behavior sets. The sequence of observations and actions that comprise each conditionally terminating sequence are shown in the tables. Each table contains all of the sequences created for one set of behavior trajectories.

Layer	Sensory variable	Bins
Layer 1	σ(etime(i1),etime(i2))	3
	passNum	3
	bblen	4
Layer 2	σ(wcp(i1),wcp(i2))	3
	passNum	3
	bblen	4
Layer 3	σ(latency(i1),latency(i2))	3
	passNum	3
	bblen	4
Layer 4	iclass conflct	2
	passNum	3
	bblen	4
Layer 5	σ(children(i1),children(i2))	3
	passNum	3
	bblen	4
Layer 6	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 7	σ(latency(i1),latency(i2))	3
	σ(etime(i1),etime(i2))	3
	passNum	3
	bblen	4
Layer 8	σ(wcp(i1),wcp(i2))	3
	σ(latency(i1),latency(i2))	3
	passNum	3
	bblen	4
Layer 9	σ(children(i1),children(i2))	3
	σ(latency(i1),latency(i2))	3
	passNum	3
	bblen	4
Layer 10	σ(etime(i1),etime(i2))	3
	σ(children(i1),children(i2))	3
	passNum	3
	bblen	4
Layer 11	σ(etime(i1),etime(i2))	3
	σ(wcp(i1),wcp(i2))	3
	passNum	3
	bblen	4
Layer 12	σ(children(i1),children(i2))	3
	σ(wcp(i1),wcp(i2))	3
	passNum	3
	bblen	4

Table B.1. Layers 1-12 of the RL scheduler's tile-coding.

Layer	Sensory variable	Bins
Layer 13	σ(`children(i1),children(i2)`)	3
	σ(`wcp(i1),wcp(i2)`)	3
	σ(`etime(i1),etime(i2)`)	3
	passNum	3
	bblen	4
Layer 14	σ(`children(i1),children(i2)`)	3
	σ(`wcp(i1),wcp(i2)`)	3
	σ(`latency(i1),latency(i2)`)	3
	passNum	3
	bblen	4
Layer 15	σ(`etime(i1),etime(i2)`)	3
	σ(`wcp(i1),wcp(i2)`)	3
	σ(`latency(i1),latency(i2)`)	3
	passNum	3
	bblen	4
Layer 16	σ(`etime(i1),etime(i2)`)	3
	σ(`wcp(i1),wcp(i2)`)	3
	σ(`latency(i1),latency(i2)`)	3
	σ(`children(i1),children(i2)`)	3
	passNum	3
	bblen	4
Layer 17	σ(`wcp(i1),wcp(i2)`)	3
	iclass conflict	2
	passNum	3
	bblen	4
Layer 18	σ(`etime(i1),etime(i2)`)	3
	iclass conflict	2
	passNum	3
	bblen	4
Layer 19	σ(`latency(i1),latency(i2)`)	3
	iclass conflict	2
	passNum	3
	bblen	4
Layer 20	σ(`children(i1),children(i2)`)	3
	iclass conflict	2
	passNum	3
	bblen	4
Layer 21	onwcp1	2
	onwcp2	2
	iclass conflict	2
	passNum	3
	bblen	4

Table B.2. Layers 13-21 of the RL scheduler's tile-coding.

Layer	Sensory variable	Bins
Layer 22	σ(latency(i1), latency(i2))	3
	σ(etime(i1), etime(i2))	3
	iclass conflict	2
	passNum	3
	bblen	4
Layer 23	σ(wcp(i1), wcp(i2))	3
	σ(latency(i1), latency(i2))	3
	iclass conflict	2
	passNum	3
	bblen	4
Layer 24	σ(children(i1), children(i2))	3
	σ(latency(i1), latency(i2))	3
	iclass conflict	2
	passNum	3
	bblen	4
Layer 25	σ(etime(i1), etime(i2))	3
	σ(children(i1), children(i2))	3
	iclass conflict	2
	passNum	3
	bblen	4
Layer 26	σ(etime(i1), etime(i2))	3
	σ(wcp(i1), wcp(i2))	3
	iclass conflict	2
	passNum	3
	bblen	4
Layer 27	σ(children(i1), children(i2))	3
	σ(wcp(i1), wcp(i2))	3
	iclass conflict	2
	passNum	3
	bblen	4
Layer 28	σ(children(i1), children(i2))	3
	σ(wcp(i1), wcp(i2))	3
	σ(etime(i1), etime(i2))	3
	iclass conflict	2
	passNum	3
	bblen	4
Layer 29	σ(children(i1), children(i2))	3
	σ(wcp(i1), wcp(i2))	3
	σ(latency(i1), latency(i2))	3
	iclass conflict	2
	passNum	3
	bblen	4

Table B.3. Layers 22-29 of the RL scheduler's tile-coding.

Layer	Sensory variable	Bins
Layer 30	σ(etime(i1),etime(i2))	3
	σ(wcp(i1),wcp(i2))	3
	σ(latency(i1),latency(i2))	3
	iclass conflict	2
	passNum	3
	bblen	4
Layer 31	σ(wcp(i1),wcp(i2))	3
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 32	σ(etime(i1),etime(i2))	3
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 33	σ(latency(i1),latency(i2))	3
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 34	σ(children(i1),children(i2))	3
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 35	σ(latency(i1),latency(i2))	3
	σ(etime(i1),etime(i2))	3
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 36	σ(wcp(i1),wcp(i2))	3
	σ(latency(i1),latency(i2))	3
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4

Table B.4. Layers 30-36 of the RL scheduler's tile-coding.

Layer	Sensory variable	Bins
Layer 37	σ(children(i1), children(i2))	3
	σ(latency(i1), latency(i2))	3
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 38	σ(etime(i1), etime(i2))	3
	σ(children(i1), children(i2))	3
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 39	σ(etime(i1), etime(i2))	3
	σ(wcp(i1), wcp(i2))	3
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 40	σ(children(i1), children(i2))	3
	σ(wcp(i1), wcp(i2))	3
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 41	σ(children(i1), children(i2))	3
	σ(wcp(i1), wcp(i2))	3
	σ(etime(i1), etime(i2))	3
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 42	σ(children(i1), children(i2))	3
	σ(wcp(i1), wcp(i2))	3
	σ(latency(i1), latency(i2))	3
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4

Table B.5. Layers 37-42 of the RL scheduler's tile-coding.

Layer	Sensory variable	Bins
Layer 43	σ(etime(i1),etime(i2))	3
	σ(wcp(i1),wcp(i2))	3
	σ(latency(i1),latency(i2))	3
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 44	σ(wcp(i1),wcp(i2))	3
	iclass conflict	2
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 45	σ(etime(i1),etime(i2))	3
	iclass conflict	2
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 46	σ(latency(i1),latency(i2))	3
	iclass conflict	2
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 47	σ(children(i1),children(i2))	3
	iclass conflict	2
	onwcp1	2
	onwcp2	2
	passNum	3
	bblen	4
Layer 48	unit1 blocked	2
	unit2 blocked	2
	σ(etime(i1),etime(i2))	3
	bblen	4

Table B.6. Layers 43-48 of the RL scheduler's tile-coding.

Layer	Sensory variable	Bins
Layer 49	unit1 blocked	2
	unit2 blocked	2
	σ(wcp(i1), wcp(i2))	3
	bblen	4
Layer 50	unit1 blocked	2
	unit2 blocked	2
	σ(children(i1), children(i2))	3
	bblen	4
Layer 51	unit1 blocked	2
	unit2 blocked	2
	σ(latency(i1), latency(i2))	3
	bblen	4

Table B.7. Layers 49-51 of the RL scheduler's tile-coding.

For these experiments, the permutation window was set to two. With this window size, the possible actions are to either swap the top instruction with the next instruction or to leave the schedule as is. These are listed as "swap" or "no swap" in the tables.

The observation at each time step used the features $\sigma(wcp1, wcp2)$, $\sigma(etime1, etime2)$, `iclass1`, and `iclass2`. The features calculated on instruction 1 refer to the top instruction in the window and instruction 2 was the following instruction. These features and the reasons for choosing them are explained in detail in Chapter 7. For the compound features calculated using $\sigma(a,b)$, the potential values are -1, 0, or 1 depending on whether a is less than b, equal to b, or greater than b, respectively. We denote the results of the comparison as $<$, $=$, and $>$ in the tables. The feature `iclass` is categorical, with each category corresponding to a functional unit in the Power PC architecture. All of the sequences were created for the Power PC 601 processor which has an integer unit (denoted IU), a floating point unit (denoted FPU), and a branching unit (denoted BPU). There is an extra unit denoted PSEUDO created to process the pseudo instructions created by JRVM.

Sequence index	Step	Action	Observations			
			σ(wcp1, wcp2)	σ(etime1, etime2)	iclass1	iclass2
1	1	swap	<	=	IU	IU
	2	no swap	=	=	IU	IU
2	1	no swap	=	=	IU	IU
	2	swap	<	=	IU	IU
3	1	swap	<	=	IU	IU
	2	no swap	=	=	IU	IU
	3	no swap	=	=	IU	IU
4	1	no swap	=	=	IU	IU
	2	swap	<	=	IU	IU
	3	no swap	=	=	IU	IU

Table B.8. Sequences generated from the BASE heuristic.

Sequence index	Step	Action	Observations			
			σ(wcp1, wcp2)	σ(etime1, etime2)	iclass1	iclass2
1	1	no swap	<	=	IU	IU
	2	swap	>	=	IU	IU
2	1	no swap	>	=	IU	IU
	2	swap	=	=	IU	IU
3	1	swap	>	=	IU	IU
	2	no swap	>	=	IU	IU
4	1	no swap	=	=	IU	IU
	2	swap	>	=	IU	IU
5	1	swap	=	=	IU	IU
	2	no swap	<	=	IU	IU
6	1	swap	>	=	IU	IU
	2	no swap	<	=	IU	IU

Table B.9. Sequences generated from the DOWNHILL heuristic.

Sequence index	Step	Action	Observations			
			σ(wcp1, wcp2)	σ(etime1, etime2)	iclass1	iclass2
1	1	swap	<	=	IU	IU
	2	no swap	=	=	IU	IU
2	1	no swap	=	=	IU	IU
	2	swap	<	=	IU	IU
3	1	no swap	=	=	IU	IU
	2	swap	<	=	IU	IU
	3	no swap	=	=	IU	IU

Table B.10. Sequences generated from the HILL heuristic.

Sequence index	Step	Action	Observations			
			σ(wcp1, wcp2)	σ(etime1, etime2)	iclass1	iclass2
1	1	swap	$<$	$=$	IU	IU
	2	swap	$=$	$=$	IU	IU
2	1	swap	$=$	$=$	IU	IU
	2	swap	$<$	$=$	IU	IU
3	1	swap	$=$	$=$	IU	IU
	2	swap	$>$	$=$	IU	IU
4	1	swap	$>$	$=$	IU	IU
	2	swap	$>$	$=$	IU	IU
5	1	swap	$=$	$=$	IU	IU
	2	swap	$=$	$=$	IU	IU
6	1	swap	$>$	$=$	IU	IU
	2	swap	$=$	$=$	IU	IU
7	1	swap	$=$	$=$	IU	IU
	2	swap	$>$	$=$	IU	IU
	3	swap	$>$	$=$	IU	IU
8	1	swap	$<$	$=$	IU	IU
	2	swap	$=$	$=$	IU	IU
	3	swap	$<$	$=$	IU	IU
9	1	swap	$<$	$=$	IU	IU
	2	swap	$=$	$=$	IU	IU
	3	swap	$=$	$=$	IU	IU
10	1	swap	$=$	$=$	IU	IU
	2	swap	$<$	$=$	IU	IU
	3	swap	$=$	$=$	IU	IU

Table B.11. Sequences generated from the FARTHEST heuristic.

Sequence index	Step	Action	Observations			
			σ(wcp1, wcp2)	σ(etime1, etime2)	iclass1	iclass2
1	1	swap	$<$	$=$	IU	IU
	2	no swap	$=$	$=$	IU	IU
2	1	swap	$<$	$=$	IU	IU
	2	swap	$=$	$=$	IU	IU
3	1	no swap	$=$	$=$	IU	IU
	2	swap	$<$	$=$	IU	IU
4	1	no swap	$=$	$=$	IU	IU
	2	swap	$=$	$=$	IU	IU
5	1	swap	$=$	$=$	IU	IU
	2	swap	$<$	$=$	IU	IU
6	1	no swap	$>$	$=$	IU	IU
	2	swap	$=$	$=$	IU	IU
7	1	swap	$=$	$=$	IU	IU
	2	no swap	$<$	$=$	IU	IU
8	1	swap	$=$	$=$	IU	IU
	2	swap	$=$	$=$	IU	IU
9	1	swap	$=$	$=$	IU	IU
	2	no swap	$=$	$=$	IU	IU
10	1	no swap	$<$	$=$	IU	IU
	2	swap	$>$	$=$	IU	IU

Table B.12. Sequences generated from the RANDOM heuristic.

Sequence index	Step	Action	Observations			
			σ(wcp1, wcp2)	σ(etime1, etime2)	iclass1	iclass2
1	1	swap	$<$	$=$	IU	IU
	2	swap	$=$	$=$	IU	IU
2	1	swap	$>$	$=$	IU	IU
	2	swap	$>$	$=$	IU	IU
3	1	swap	$=$	$=$	IU	IU
	2	swap	$<$	$=$	IU	IU
4	1	swap	$<$	$=$	PSEUDO	PSEUDO
	2	swap	$<$	$=$	PSEUDO	PSEUDO
5	1	swap	$<$	$<$	PSEUDO	PSEUDO
	2	swap	$<$	$<$	PSEUDO	PSEUDO
6	1	swap	$<$	$=$	PSEUDO	PSEUDO
	2	swap	$<$	$<$	PSEUDO	PSEUDO
7	1	swap	$<$	$<$	PSEUDO	PSEUDO
	2	swap	$<$	$<$	PSEUDO	PSEUDO
	3	swap	$<$	$<$	PSEUDO	PSEUDO
8	1	swap	$<$	$=$	PSEUDO	PSEUDO
	2	swap	$<$	$=$	PSEUDO	PSEUDO
	3	swap	$<$	$<$	PSEUDO	PSEUDO
9	1	swap	$<$	$<$	PSEUDO	PSEUDO
	2	swap	$<$	$<$	PSEUDO	PSEUDO
	3	swap	$<$	$<$	PSEUDO	PSEUDO
	4	swap	$<$	$<$	PSEUDO	PSEUDO

Table B.13. Sequences generated from the COMPRESS heuristic.

Sequence index	Step	Action	Observations			
			σ(wcp1, wcp2)	σ(etime1, etime2)	iclass1	iclass2
1	1	swap	$<$	$=$	PSEUDO	PSEUDO
	2	swap	$<$	$=$	PSEUDO	PSEUDO
2	1	swap	$<$	$=$	IU	IU
	2	swap	$=$	$=$	IU	IU
3	1	swap	$>$	$=$	IU	IU
	2	swap	$>$	$=$	IU	IU
4	1	swap	$<$	$=$	IU	IU
	2	no swap	$=$	$=$	IU	IU
5	1	swap	$<$	$=$	IU	IU
	2	swap	$=$	$>$	IU	IU
6	1	no swap	$=$	$=$	IU	IU
	2	swap	$<$	$=$	IU	IU

Table B.14. Sequences generated from the RAYTRACE heuristic.

Sequence index	Step	Action	Observations			
			σ(wcp1, wcp2)	σ(etime1, etime2)	iclass1	iclass2
1	1	swap	$<$	$<$	PSEUDO	PSEUDO
	2	swap	$<$	$<$	PSEUDO	PSEUDO
2	1	swap	$<$	$=$	PSEUDO	PSEUDO
	2	swap	$<$	$<$	PSEUDO	PSEUDO
3	1	swap	$<$	$=$	PSEUDO	PSEUDO
	2	swap	$<$	$=$	PSEUDO	PSEUDO
4	1	swap	$<$	$<$	PSEUDO	PSEUDO
	2	swap	$<$	$<$	PSEUDO	PSEUDO
	3	swap	$<$	$<$	PSEUDO	PSEUDO
5	1	swap	$<$	$=$	PSEUDO	PSEUDO
	2	swap	$<$	$<$	PSEUDO	PSEUDO
	3	swap	$<$	$<$	PSEUDO	PSEUDO
6	1	swap	$<$	$=$	PSEUDO	PSEUDO
	2	swap	$<$	$=$	PSEUDO	PSEUDO
	3	swap	$<$	$<$	PSEUDO	PSEUDO
7	1	swap	$<$	$<$	PSEUDO	PSEUDO
	2	swap	$<$	$<$	PSEUDO	PSEUDO
	3	swap	$<$	$<$	PSEUDO	PSEUDO
	4	swap	$<$	$<$	PSEUDO	PSEUDO
8	1	swap	$<$	$=$	PSEUDO	PSEUDO
	2	swap	$<$	$<$	PSEUDO	PSEUDO
	3	swap	$<$	$<$	PSEUDO	PSEUDO
	4	swap	$<$	$<$	PSEUDO	PSEUDO
9	1	swap	$<$	$=$	PSEUDO	PSEUDO
	2	swap	$<$	$=$	PSEUDO	PSEUDO
	3	swap	$<$	$<$	PSEUDO	PSEUDO
	4	swap	$<$	$<$	PSEUDO	PSEUDO
10	1	swap	$<$	$<$	PSEUDO	PSEUDO
	2	swap	$<$	$<$	PSEUDO	PSEUDO
	3	swap	$<$	$<$	PSEUDO	PSEUDO
	4	swap	$<$	$<$	PSEUDO	PSEUDO
	5	swap	$<$	$<$	PSEUDO	PSEUDO

Table B.15. Sequences generated from the CROSS heuristic.

Benchmark	Heuristics					
	SUB-OPTIMAL	FARTHEST	RANDOM	HILL	BASE-HILL	OPTIMAL
Compress	1.09	1.04	1.02 ± 0.01	0.99	0.96	0.96
Jess	1.04	1.02	1.01 ± 0.00	0.99	0.98	0.98
Db	1.04	1.03	1.01 ± 0.01	1.00	0.99	0.99
Javac	1.05	1.01	1.01 ± 0.00	0.99	0.99	0.99
Mpegaudio	1.03	1.01	0.98 ± 0.01	1.00	0.90	0.90
Raytrace	1.09	1.04	1.02 ± 0.00	0.96	0.96	0.96
Jack	1.07	1.02	1.01 ± 0.01	1.00	1.00	0.98

Table B.16. Performance ratios for each of the heuristics on the Power PC 601 processor.

B.3 Detailed Results for Each Instruction Scheduling Experiment

Table B.16 gives the performance ratios for each of the heuristic schedulers on each of the seven benchmarks. Each of these heuristics is run without any sequences. All of the heuristics except RANDOM are deterministic and the numbers for these are from one run. For RANDOM, we report the average and standard deviation over 30 runs, each starting with a different random seed. The ratios are all calculated as described by Equation 7.1.

Table B.17 shows the differences between the mean performance ratio of the random scheduler with and without sequences. Each of the means is calculated over 30 different runs. A positive number in the table means that the random scheduler using the specified set of sequences performed better than the random scheduler without sequences. Blank entries performed the same as the random scheduler, on average.

Table B.18 shows the percent improvement toward the estimated optimal scheduler of the hill climbing scheduler using sequences. The sequences used for these experiments are noted in each row of the table. Each behavior set was generated on the Power PC 601 processor. These results were also generated on the 601 processor. Each of the percent improvement numbers was calculated according to Equation 7.2.

Table B.19 details the improvement of the hill climber with sequences toward the estimated optimal scheduler on the Power PC 603 processor. The sequences used in this experiment were generated on the Power PC 601 processor.

Benchmark	SEQ-DOWNHILL	SEQ-FARTHEST	SEQ-RANDOM
Compress		0.02	0.01
Jess	-0.01		
Db	-0.01	-0.01	
Javac	-0.01		
Mpegaudio		0.02	0.02
Raytrace	-0.01		
Jack			

Benchmark	SEQ-BASE	SEQ-HILL	SEQ-COMPRESS	SEQ-RAYTRACE	SEQ-CROSS
Compress	0.01	0.01	0.02	0.01	0.01
Jess	0.01	0.01			
Db	0.01		-0.01		
Javac	0.01	0.01			
Mpegaudio	0.02	0.02	0.03	0.01	0.02
Raytrace	0.01	0.01		0.01	
Jack					

Table B.17. Differences in the mean performance ratio of RANDOM without sequences to RANDOM with the specified set of sequences. Numbers less than 0 represent a worse performance than scheduling randomly without sequences and numbers greater than 0 represent a better performance.

Sequences	Benchmarks						
SEQ-	Compress	Jess	Db	Javac	Mpegaudio	Raytrace	Jack
FARTHEST	52.49	40.32	69.86	43.85	33.22	90.30	3.37
RANDOM	37.83	40.29	69.86	41.11	28.01	34.17	2.71
COMPRESS	8.11	17.07	46.31	39.41	21.86	25.53	0.34
RAYTRACE	15.90	11.16	46.30	37.16	17.58	2.52	0.19
DOWNHILL	13.43	-0.35	-0.01	8.36	17.25	0.09	0.01
HILL	11.84	-0.37	-0.01	6.83	17.25	-10.11	-0.15
BASE	11.84	-0.37	-0.01	6.83	17.25	-10.11	-0.15
CROSS		-0.35	-0.01		17.25	0.09	

Table B.18. Percent relative improvement toward an estimate of optimal of the hill climbing scheduler with sequences on the Power PC 601 processor.

Sequences	**Benchmarks**						
SEQ-	`Compress`	`Jess`	`Db`	`Javac`	`Mpegaudio`	`Raytrace`	`Jack`
FARTHEST	38.74	8.90	-0.17	16.79	18.86	61.50	7.27
RANDOM	38.74	2.03	-0.17	13.43	18.86	59.62	6.68
COMPRESS	9.36	-0.74	-0.17	4.31	13.30	9.18	
RAYTRACE		-0.74	-0.17	0.05	0.09	7.01	
DOWNHILL		-0.74	-0.17				
HILL		-0.74	-0.17				
BASE		-0.74	-0.17				
CROSS		-0.74	-0.17				

Table B.19. Percent relative improvement of the hill climbing scheduler toward the estimated optimal scheduler using the sequences generated on the Power PC 601 processor on the Power PC 603 processor.

BIBLIOGRAPHY

Albus, J. S. (1971). A theory of cerebellar function. *Mathematical Biosciences*, 10:25–61.

Alpern, B., Attanasio, C. R., Barton, J. J., Burke, M. G., P.Cheng, Choi, J.-D., Cocchi, A., Fink, S. J., Grove, D., Hind, M., Hummel, S. F., Lieber, D., Litvinov, V., Mergen, M. F., Ngo, T., Russell, J. R., Sarkar, V., Serrano, M. J., Shepherd, J. C., Smith, S. E., Sreedhar, V. C., Srinivasan, H., and Whaley, J. (2000). The Jalapeño virtual machine. *IBM System Journal*, 39(1).

Amarel, S. (1968). On representations of problems of reasoning about actions. In Michie, D., editor, *Machine Intelligence 3*, volume 3, pages 131–171. Elsevier, North Holland, Amsterdam, London, New York.

Andre, D. and Russell, S. J. (2001). Programmable reinforcement learning agents. In *Proceedings of Advances in Neural Information Processing Systems 13*, pages 1019–1025. MIT Press.

Anzai, Y. and Simon, H. A. (1979). The theory of learning by doing. *Psychological Review*, 86(2):124–140.

Ballard, D. H. and Brown, C. M. (1982). *Computer Vision*. Prentice-Hall, Inc., Englewood Cliffs, NJ.

Bernstein, D. S. (1999). Reusing old policies to accelerate learning on new MDPs. Technical Report UM-CS-1999-026, Department of Computer Science, University of Massachusetts, Amherst, Amherst, MA.

Boutilier, C., Dean, T., and Hanks, S. (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence*, 11:1–94.

Boutilier, C., Dearden, R., and Goldszmidt, M. (1995). Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1550–1556.

Bradtke, S. J. and Duff, M. O. (1995). Reinforcement learning methods for continuous-time Markov decision problems. In *Advances in Neural Information Processing Systems 7*. MIT Press.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23. also MIT AI Memo 864, September 1985.

Brooks, R. A. (1990). Elephants don't play chess. *Robotics and Autonomous Systems*, 6:3–15.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. The MIT Press, Cambridge, MA.

Craig, J. J. (1989). *Introduction to Robotics: Mechanics and Control*. Addison-Wesley, second edition.

Dayan, P. and Hinton, G. E. (1993). Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5*, pages 271–278. Morgan Kaufmann.

Dean, T. and Lin, S.-H. (1995). Decomposition techniques for planning in stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1121–1127, Montreal, CA.

Dietterich, T. G. (1998). Hierarchical reinforcement learning with the MAXQ value function decomposition. In *Proceedings of the 15th International Conference on Machine Learning ICML'98*, San Mateo, CA. Morgan Kaufmann.

Dietterich, T. G., Lathrop, R. H., and Lozano-Perez, T. (1997). Solving the multiple-instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89(1-2):31–71.

Digney, B. (1996). Emergent hierarchical control structures: Learning reactive/hierarchical relationships in reinforcement environments. In Maes, P. and Mataric, M., editors, *From Animals to Animats 4: The Fourth Conference on the Simulation of Adaptive Behavior SAB 96*. MIT Press/Bradford Books.

Digney, B. (1998). Learning hierarchical control structure for multiple tasks and changing environments. In *From Animals to Animats 5: The Fifth Conference on the Simulation of Adaptive Behavior: SAB 98*.

Drummond, C. (1998). Composing functions to speed up reinforcement learning in a changing world. In *European Conference on Machine Learning*, pages 370–381.

Fikes, R. E., Hart, P. E., and Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288.

Gullapalli, V. (1992). *Reinforcement Learning and its Application to Control*. PhD thesis, University of Massachusetts, Amherst.

Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge.

Hansen, E. A., Barto, A. G., and Zilberstein, S. (1997). Reinforcement learning for mixed open-loop and closed-loop control. In Mozer, M. C., Jordan, M. I., and Petsche, T., editors, *Advances in Neural Information Processing Systems*, volume 9, pages 1026–1032. The MIT Press.

Hauskrecht, M., Meuleau, N., Boutilier, C., Kaelbling, L. P., and Dean, T. (1998). Hierarchical solution of Markov decision processes using macro-actions. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence*, pages 220–229.

Hough, P. (1962). Method and means for recognizing complex patterns. U.S. Patent 3,069,654.

Huber, M. (2000). *A Hybrid Architecture for Adaptive Robot Control*. PhD thesis, University of Massachusetts, Amherst.

Huber, M. and Grupen, R. A. (1997a). A feedback control structure for on-line learning tasks. *Robots and Autonomous Systems*, 22(3-4):303–315.

Huber, M. and Grupen, R. A. (1997b). Learning to coordinate controllers - reinforcement learning on a control basis. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1366–1371.

Huber, M. and Grupen, R. A. (1998). Learning robot control - using control policies as abstract actions. Proceedings of the NIPS '98 Workshop on Abstraction and Hierarchy in Reinforcement Learning.

Huber, M. and Grupen, R. A. (1999). A hybrid architecture for learning robot control tasks. In *Proceedings of the 1999 AAAI Spring Symposium Series: Hybrid Systems and AI: Modeling, Analysis and Control of Discrete + Continuous Systems*, Stanford University, CA.

Huber, M., MacDonald, W. S., and Grupen, R. A. (1996). A control basis for multilegged walking. In *Proceedings of the IEEE Conference on Robotics and Automation*, volume 4, pages 2988–2993, Minneapolis, MN.

Hui, L. C. K. (1992). Color set size problem with applications to string matching. In Apostolico, A., Crochemore, M., Galil, Z., and Manber, U., editors, *Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, volume 644 of *Lecture Notes in Computer Science*, pages 227–240. Springer Verlag, Berlin, Heidelberg, New York.

Iba, G. A. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3:285–317.

Kaelbling, L. (1993). Hierarchical learning in stochastic domains: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 167–173. Morgan Kaufmann.

Knoblock, C. A. (1990). Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, volume 2, pages 923–928, Boston, MA. MIT Press.

Knoblock, C. A. (1991). *Automatically Generating Abstractions for Problem Solving*. PhD thesis, School of Computer Science, Carnegie Mellon University. Available as Technical report CMU-CS-91-120.

Korf, R. E. (1985). Macro-operators: A weak method for learning. *Artificial Intelligence*, 26:35–77.

Laird, J. E., Rosenbloom, P. S., and Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46.

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321.

Lin, L. J. (1993a). *Reinforcement Learning for Robots using Neural Networks*. PhD thesis, Carnegie Mellon University, School of Computer Science.

Lin, L.-J. (1993b). Scaling up reinforcement learning for robot control. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 182–189. Morgan Kaufmann.

Mahadevan, S. and Connell, J. (1992). Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55:311–365.

Makar, R., Mahadevan, S., and Ghavamzadeh, M. (2001). Hierarchical multi-agent reinforcement learning. In *Proceedings of the Fifth International Conference on Autonomous Agents*, Montreal, Canada.

Maron, O. (1998). *Learning from Ambiguity*. PhD thesis, Massachusetts Institute of Technology.

Maron, O. and Lozano-Pérez, T. (1998). A framework for multiple-instance learning. In Jordan, M. I., Kearns, M. J., and Solla, S. A., editors, *Advances in Neural Information Processing Systems 10*, pages 570–576, Cambridge, Massachusetts. MIT Press.

McCallum, A. K. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, Rochester.

McGovern, A. (1998a). acQuire-macros: An algorithm for automatically learning macro-actions. Proceedings of the NIPS '98 Workshop on Abstraction and Hierarchy in Reinforcement Learning.

McGovern, A. (1998b). Roles of macro-actions in accelerating reinforcement learning. Master's thesis, University of Massachusetts, Amherst. Also University of Massachusetts, Amherst Technical Report 98-70.

McGovern, A. and Barto, A. G. (2001). Linear discriminant diverse density for automatic discovery of subgoals in reinforcement learning. Poster presentation at the Workshop on Hierarchy and Memory in Reinforcement Learning at the 18th International Conference on Machine Learning.

McGovern, A., Moss, E., and Barto, A. G. (2002). Building a basic block instruction scheduler with reinforcement learning and rollouts. *Machine Learning*, 49(2/3):141–160.

McGovern, A., Sutton, R. S., and Fagg, A. H. (1997). Roles of macro-actions in accelerating reinforcement learning. In *Proceedings of the 1997 Grace Hopper Celebration of Women in Computing*, pages 13–18.

Minton, S. (1988). *Learning Search Control Knowledge: An Explanation-based Approach*. Kluwer Academic Publishers.

Moore, A. W. (1994). The parti-game algorithm for variable resolution reinforcement learning in multidimensional spaces. In Cohen, J. D., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems: Proceedings of the 1993 Conference*, pages 711–718, San Francisco, CA. Morgan Kaufmann.

Moore, A. W., Baird, L. C., and Kaelbling, L. (1999). Multi-value-functions: Efficient automatic action hierarchies for multiple goal MDPs. In Dean, T., editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI99)*, volume 2, pages 1316–1321, San Francisco, CA. Morgan Kauffman Publishers, Inc.

Newell, A., Shaw, J. C., and Simon, H. A. (1963). GPS, a program that simulates human thought. In Feigenbaum, E. A. and Feldman, J., editors, *Computers and Thought*, pages 279–293. McGraw-Hill, New York.

Oates, T. (1999). Identifying distinctive subsequences in multivariate time series by clustering. In *Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining*.

Oates, T., Jensen, D., and Cohen, P. R. (1998). Discovering rules for clustering and predicting asynchronous events. In *Predicting the Future: AI Approaches to Time Series Workshop, AAAI-98*, pages 73–79.

Parr, R. (1998). *Hierarchical Control and Learning for Markov Decision Processes*. PhD thesis, University of California at Berkeley.

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

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, San Mateo, California.

Precup, D. (2000). *Temporal Abstraction in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst.

Precup, D. and Sutton, R. S. (1997). Multi-time models for temporally abstract planning. In *Proceedings of Advances in Neural Information Processing Systems 10*. MIT Press.

Precup, D., Sutton, R. S., and Singh, S. (1998). Theoretical results on reinforcement learning with temporally abstract behaviors. In *Proceedings of the Tenth European Conference on Machine Learning, ECML'98*. Springer-Verlag.

Prieditis, A. E. (1993). Machine discovery of effective admissible heuristics. *Machine Learning*, 12:117–141.

Proebsting, T. (1998). Least-cost instruction selection in DAGs is NP-Complete. http://www.research.microsoft.com/∼toddpro/papers/proof.htm.

Ram, A. and Santamaría, J. (1997). Continuous case based reasoning. *Artificial Intelligence*, 90:25–77.

Rosenstein, M. T. and Barto, A. G. (2001). Robot weightlifting by direct policy search. In Nebel, B., editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, volume 2, pages 839–844, San Francisco. Morgan Kaufmann.

Rosenstein, M. T. and Cohen, P. R. (1999). Continuous categories for a mobile robot. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*.

Ryan, M. and Pendrith, M. (1998). Rl-tops: An architecture for modularity and re-use in reinforcement learning. In *Proceedings of the 15th International Conference on Machine Learning ICML'98*, San Mateo, CA. Morgan Kaufmann.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135.

Singh, S. P. (1991). Transfer of learning across compositions of sequential tasks. In Birnbaum, L. A. and Collins, G. C., editors, *Proceedings of the Eighth International Workshop on Machine Learning*, pages 348–352, San Mateo, CA. Morgan Kaufmann.

Singh, S. P. (1992a). The efficient learning of multiple task sequences. In Moody, J., Hanson, S., and Lippman, R., editors, *Advances in Neural Information Processing Systems: Proceedings of the 1991 Conference*, pages 251–258, San Mateo, CA. Morgan Kaufmann.

Singh, S. P. (1992b). Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 202–207, Menlo Park, CA. AAAI Press/MIT Press.

Singh, S. P. (1992c). Transfer of learning by composing solutions for elemental sequential tasks. *Machine Learning*, 8:323–339.

Singh, S. P. (1994). *Learning to Solve Markovian Decision Processes*. PhD thesis, University of Massachusetts, Amherst.

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44.

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

Sutton, R. S., Precup, D., and Singh, S. (1998). Between MDPs and Semi-MDPs: learning, planning, and representing knowledge at multiple temporal scales. Technical Report 98-74, University of Massachusetts, Amherst.

Sutton, R. S., Precup, D., and Singh, S. (1999). Between MDPs and Semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211.

Theocharous, G. and Mahadevan, S. (2002). Approximate planning with hierarchical partially observable markov decision processes for robot navigation. In *Proceedings of the 2002 IEEE Conference on Robotics and Automation (ICRA)*, Washington, DC.

Thrun, S., Fox, D., Burgard, W., and Dellaert, F. (2000). Robust Monte Carlo localization for mobile robots. *Artificial Intelligence*, 101:99–141.

Thrun, S. B. and Schwartz, A. (1995). Finding structure in reinforcement learning. In G. Tesauro, D. Touretzky, T. L., editor, *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference*, pages 385–392, San Mateo, CA. Morgan Kaufmann.

Uther, W. T. B. (1998). Automated acquisition of hierarchical decomposition graphs for continuous state space reinforcement learning. Thesis Proposal. Personal Communication.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards*. PhD thesis, Cambridge University, Cambridge, England.

Zaki, M. J. (1998). Efficient enumeration of frequent sequences. In *Proceedings of the 7th International Conference on Information and Knowledge Management*.

Zaki, M. J. (2001). Spade: An efficient algorithm for mining frequent sequences. *Machine Learning Journal*, 42(1/2):31–60. Special issue on unsupervised learning.